A Web Application for Tomato Leaf Disease Diagnosis: A Comparative Analysis of Random Forest and Customized CNN Models

A Step Forward in Developing Practical Solutions to Improve the Lives of Local farmers

Oboni Anower

School of Computing and Mathematical Sciences
Birkbeck, University of London

January 2025

Declaration

This report is the result of my own work except where explicitly indicated in the text. I give my permission for it to be submitted to the TURNITIN Plagiarism Detection Service. I have read and understood the sections on plagiarism in the College website and the Policy and Procedures on academic integrity. The report may be freely copied and distributed provided the source is explicitly acknowledged.

Signature: obonianower

Date:

30/01/2025

Abstract

Tomato is one of the most popular fruits that can also be eaten as a vegetable. The plants are relatively easy to cultivate, however they come with various diseases that can disrupt production efficiency on a large scale. Some commonly seen diseases tomato leaves face are early and late blights. Many leaves also develop bacterial, target and septoria leaf spots in rainy weather. Oftentimes, mosaic and yellow leaf curl viruses infect the leaves in warm and humid conditions. Some other tomato plant diseases include spider mites, which is usually caused by insecticide sprays. Ever since the development of machine learning and advancement into deep learning, disease classification has become an important research concept in precision agriculture. To enable early detection of plant diseases, researchers have been utilizing advanced computing technologies such as computer vision and deep learning architectures to develop disease recognition systems for early detection. This research proposed a comparative analysis between two deep learning architectures to determine a classification model with reliable performance. The research was conducted using Tomato leaf images from PlantVillage dataset to classify from ten different health categories. The dataset was pre-processed to enhance the images and address data imbalance. The proposal leveraged a neural network VGG16's five convolutional blocks as a feature extractor using transfer learning and built Random Forest, a machine learning algorithm as a classifier. Similarly, a custom Neural Network architecture was designed and stacked on top of the feature extractor to build a second classifier. The classifiers were optimized using hyperparameter tuning techniques and the best parameters were used to strategize the experimental scenarios. Each classifier was trained with different hyperparameter combinations, and the models were tested against unseen data to evaluate their generalization abilities. The tests helped to select the optimal model from both classifier types based on their performance accuracies. Model evaluations and performance metrics showed that the custom CNN model correctly predicted true disease instances with an accuracy of 85%, whereas the Random Forest achieved only 80%. Because CNN outperformed, the model was finalised to be deployed into a web interface to allow users to upload infected images and determine plant health. The ultimate motivation behind this research was to develop a reliable diagnostic tool for farmers that can detect tomato leaf disease at an early stage and mitigate financial loss.

Contents

Declaration I

Abstract II

Contents III

List of Figures and Tables V

Glossary VI

List of Abbreviations VIII

Acknowledgements IX

1 Introduction 1

2 Aims and Objectives 3

3 Literature Reviews 5

4 Methodology and Methods 6

4.1 Data Acquisition 7

4.2 Data Division 8

4.3 Data Preprocessing 9

4.4 Feature Extraction 10

4.5 Class Balancing 12

4.6 Dimensionality Reduction for Machine Learning 14

4.7 Model Architectures 15

4.7.1 Machine Learning Algorithm 15

4.7.2 CNN Algorithm 16

4.8 Evaluation Metrics 17

4.9 Hardware and Software Requirements 18

5 Implementations and Training Process 19

5.1 Random Forest 19

5.1.1 Base model 19

5.1.2 Hyperparameter Tuning with Random Search 19
5.1.3 Experimental Hyperparameter Tuning 20
5.2 Custom CNN 20
5.2.1 Hyperparameter Tuning 20
5.2.2 Fine-tuning Experiments 21
6 Experimental Analysis 22
6.1 Random Forest 22
6.2 Custom CNN 24
7 Classification Results and Discussion 28
7.1 Model Comparison 28
7.2 Comparison with Existing Work 30
8 Web Application 31
9 Conclusions 32
References 34
Appendix A 38
Appendix B 39
Appendix C 40
Appendix D 41
Appendix E 42
Appendix F 43
Appendix G 44
Appendix H 45
Appendix I 46
Appendix J 47
Appendix K 48
Appendix L 49
Appendix M 50

Appendix N 51

List of Figures and Tables

Figure 1: Project Workflow 6

Figure 2: Randomly generated phenotypes of tomato plants from each class directory of training data 7

Figure 3: Visual representation of VGG16 architecture for extracting feature maps 10

Figure 4: Class Imbalance 12

Figure 5: PCA components for 95% variance 14

Figure 6: Training and validation accuracy (left) and loss (right) per epoch 27

Figure 7: Model Performance Breakdown by Class 28

Figure 8: Classification report 29

Figure 9: Architecture of the proposed custom CNN classifier 30

Figure 10: Confusion Matrix of Custom CNN Model (left) and Random Forest Classifier (right) 48

Table 1: Class names and number of images in each class 7

Table 2: Search space for Hyperparameter tuning using Random Search for Random Forest Classifier 19

Table 3: Hyperparameter space for Custom CNN using Random Search 21

Table 4: Parameters of Default Random Search Classifier (middle column); Parameters found after Random search with given search-space (right column) 22

Table 5: Experimental results for Random Forest Hyperparameter combinations 23

Table 6: Experimental Trials 1-12 for the Custom CNN 24

Table 7: Experimental Trials 13-24 for the Custom CNN 25

Table 8: Experimental Trials 25 & 26 to improve training accuracy with 0% dropout 26

Table 9: Hyperparameters of the finalized models 27

Table 10: Comparison of the best CNN and ML models 28

Table 11: Test results of each category for Random Forest (left) and custom CNN model (right) 29

Table 12: Performance comparison of with existing related research using PlantVillage dataset 30

Glossary

Term	Description
Accuracy	Percentage of correct predictions made by the model.
Algorithm	A method, function, or series of instructions used to generate a machine learning model. Examples include linear regression, decision trees, support vector machines, and neural networks.
Classification	Predicting a categorical output. Multi-class classification predicts one of multiple possible outcomes (e.g. is this a photo of a cat, dog, horse or human?)
Confusion Matrix	Table that describes the performance of a classification model by grouping predictions into 4 categories.
Convergence	A state reached during the training of a model when the loss changes very little between each iteration.
Deep Learning	Deep Learning is derived from a machine learning algorithm called perceptron or multi-layer perceptron that is gaining more and more attention nowadays because of its success in different fields like, computer vision to signal processing and medical diagnosis to self-driving cars. Like other AI algorithms, deep learning is based on decades of research. Nowadays, we have more and more data and cheap computing power that makes this algorithm really powerful in achieving state of the art accuracy. In the modern world this algorithm is known as an artificial neural network. Deep learning is much more accurate and robust compared to traditional artificial neural networks. But it is highly influenced by machine learning's neural network and perceptron networks.
Dimension	Dimension for machine learning and data science is different from physics. Here, dimension of data means how many features you have in your data ocean(data-set). e.g. in case of object detection application, flatten image size and colour channel(e.g. 28*28*3) is a feature of the input set. In case of house price prediction (maybe) house size is the data-set so we call it 1 dimensional data.
Epoch	An epoch describes the number of times the algorithm sees the entire data set.
Feature	With respect to a dataset, a feature represents an attribute and value combination. Colour is an attribute. "Colour is blue" is a feature. In Excel terms, features are similar to cells. The term feature has other definitions in different contexts.
Generalization gap	The difference between Validation accuracy and test accuracy.
Hyperparameters	Hyperparameters are higher-level properties of a model such as how fast it can learn (learning rate) or complexity of a model. The depth of trees in a Decision Tree or number of hidden layers in a Neural Networks are examples of hyper parameters.
Instance	A data point, row, or sample in a dataset. Another term for observation.

Label	The "answer" portion of an observation in supervised learning. For example, in a dataset used to classify flowers into different species, the features might include the petal length and petal width, while the label would be the flower's species.
Learning Rate	The size of the update steps to take during optimization loops like Gradient Descent. With a high learning rate we can cover more ground each step, but we risk overshooting the lowest point since the slope of the hill is constantly changing. With a very low learning rate, we can confidently move in the direction of the negative gradient since we are recalculating it so frequently. A low learning rate is more precise, but calculating the gradient is time-consuming, so it will take us a very long time to get to the bottom.
Loss	Loss = true value(from data-set)- predicted value(from ML-model) The lower the loss, the better a model (unless the model has over-fitted to the training data). The loss is calculated on training and validation and its interpretation is how well the model is doing for these two sets. Unlike accuracy, loss is not a percentage. It is a summation of the errors made for each example in training or validation sets.
Machine Learning	Mitchell (1997) provides a succinct definition: "A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E." In simple language machine learning is a field in which human made algorithms have an ability learn by itself or predict future for unseen data.
Model	A data structure that stores a representation of a dataset (weights and biases). Models are created/learned when you train an algorithm on a dataset.
Neural Networks	Neural Networks are mathematical algorithms modelled after the brain's architecture, designed to recognize patterns and relationships in data.
Normalization	Restriction of the values of weights in regression to avoid overfitting and improving computation speed.
Noise	Any irrelevant information or randomness in a dataset which obscures the underlying pattern.
Overfitting	Overfitting occurs when your model learns the training data too well and incorporates details and noise specific to your dataset. You can tell a model is overfitting when it performs great on your training/validation set, but poorly on your test set (or new real-world data).
Parameters	Parameters are properties of training data learned by training a machine learning model or classifier. They are adjusted using optimization algorithms and unique to each experiment. Examples of parameters include weights in an artificial neural network, support vectors in a support vector machine, coefficients in a linear or logistic regression.
Regularization	Regularization is a technique utilized to combat the overfitting problem. This is achieved by adding a complexity term to the loss function that gives a bigger loss for more complex models.
Supervised Learning	Training a model using a labelled dataset.
Test Set	A set of observations used at the end of model training and validation to assess the predictive power of your model.
Training Set	A set of observations used to generate machine learning models.

Transfer Learning	A machine learning method where a model developed for a task is reused as the starting point for a model on a second task. In transfer learning, we take the pre-trained weights of an already trained model (one that has been trained on millions of images belonging to 1000's of classes, on several high power GPU's for several days) and use these already learned features to predict new classes.
Underfitting	Underfitting occurs when your model over-generalizes and fails to incorporate relevant variations in your data that would give your model more predictive power. You can tell a model is underfitting when it performs poorly on both training and test sets.
Validation Set	A set of observations used during model training to provide feedback on how well the current parameters generalize beyond the training set. If training error decreases but validation error increases, your model is likely overfitting, and you should pause training.
Variance	Explains how tightly packed your predictions for a particular observation are relative to each other. Low variance suggests your model is internally consistent, with predictions varying little from each other after every iteration. High variance (with low bias) suggests your model may be overfitting and reading too deeply into the noise found in every training set.

(Joseph, 2022)

List of Abbreviations

Abbreviations	Expanded Form
ML	Machine Learning
DL	Deep learning
CNN	Convolutional Neural Network
1-D	1-Dimensional
2-D	2-Dimensional
PCA	Principal Component Analysis
RF	Random Forest
SVM	Support Vector machine
FC	Fully connected (layer)
LR	Learning Rate

Acknowledgements

First and foremost, I would like to express my gratitude to my supervisor, Professor George D. Magoulas for their unwavering guidance and mentorship throughout this research. They always responded to my concerns timely, provided me with invaluable feedback, and always advised me to bring the best out of my work.

I would like to appreciate my friends Aneeza Z. A., Imran K., Neha V., and Shabnam K., for their constant emotional support, which helped me get through the hard days.

I am grateful to my dad and mom for doing the best they could.

Lastly, I thank God for always making a way for me.

1 Introduction

In tropical countries like Bangladesh, tomato farming has not only created employment prospects for the educated youths of the country, but also has the potential to change the socio-economic condition of the poorer population. Tomato, or *Solanum lycopersicum L.* is a highly profitable seasonal crop in the region as farmers can harvest them multiple times during the season. However, these plants are vulnerable to a handful of insect pests and plant pathogens (Depenbusch, et al., 2023), causing a constant battle for farmers. Some commonly seen tomato leaf diseases are types of blight. Early Blight are dark lesions with yellow surroundings on leaves and can occur at any stage of the growing season, getting worse with higher temperatures. Late blight usually occurs in persistent humid conditions with mild temperature. The infection looks greenish-black which turns brown over time. Septoria leaf spot is a devastating tomato leaf disease caused by a fungal pathogen called *Septoria lycopersici*, making tiny, grey and round spots with dark borders on the foliage. Wet and rainy weather can cause an outbreak of Bacterial spots, showing small, water soaked circular spots that are brownish. Tobacco mosaic virus infects tomato leaves and makes them coiled, crumpled and smaller than their regular size. Insects like whitefly can transmit Leaf Curl Virus which makes the leaves go yellowish and curly, leading to slow growth. Spider Mites can infect the back of the leaves under extreme temperature with dry weather (Singh, et al., n.d.). Many other diseases due to environmental factors and even insecticide can cause plant health to deteriorate. For a country where agriculture contributes to 20% of the GDP and accounts for about 47% of the labour force (Rahman & Hossain, 2014), crop diseases can pose detrimental impacts on its economy and food safety. If not promptly identified and controlled in time, infections can spread rapidly and can cause substantial cutbacks in yield. This can lead to detrimental commercial impact on local farmers whose livelihood depends on these crops. Habitually, local farmers rely on their years of manual field experience to identify crop deterioration because of limited access to agricultural pathologists in the remotest part of the region (Anower, 2024). Due to the lack of knowledge and information about the disease, farmers often spend large amounts of chemical pesticides to protect their crops, which is economically inconvenient as well as often unhealthy for human consumption. But the issue can be prevented if farmers can find a solution to identify and treat these diseases effectively at an early stage. Such a solution has been offered by Islam, et al. (2023) by creating a web application that exploited advanced computing to fast track the disease recognition mechanism in real time. Similarly, this research aimed to develop a webpage that can utilize a robust Deep-Learning based image classifier to predict tomato leaf diseases with high efficiency. The goal was to explore Machine Learning (ML) and Convolutional Neural Network (CNN) architectures to analyse their comparative performance before choosing the most competent one as the final classifier. The relevance of this research is to construct a computer aided monitoring and diagnostic tool that can assist farmers take preventive actions in advance without spending too many resources, which can not only lead to better production yields, but also benefit them financially. Moreover, this research plans to present an analysis of the final performance with existing benchmark methods to identify its weaknesses for future improvements.

With the surge of artificial intelligence and high availability of data over the years, agricultural researchers have been making promising use of cutting-edge computational powers to create automated disease detection systems for improved farming practices. The application of ML in plant disease identification has been the subject of interest over the last couple of decades due to its reliability in predictive analysis. CNN is recognized as the most powerful method for image classification tasks. Computers utilize its layered structures to extract features from images to classify them precisely. Many CNN architectures have provenly achieved very high accuracies for agricultural image classification tasks, for instance the classification of paddy leaf diseases using MobileNet (Masykur, et al., 2022). For

image classification, features are needed to be extracted from the images first. To examine the efficiency of a strong neural network structure in classifying tomato plant diseases, Banerjee, et al. (2024) attempted a unified approach by combining an ML model with a custom CNN of four convolutional layers. The model leveraged CNN's ability to capture complex hierarchical patterns from raw images using reverse propagation and slope descent techniques. The combined features in the dense layer were fed to a Random Forest classifier to predict the tomato diseases. The predictions were then passed to the CNN's output dense layer for the final classification. This research was useful for preventing overfits and improving generalization, as the model achieved an overall accuracy of 92.84%, demonstrating its reliability.

However, training a CNN model from the ground up can be computationally intensive because its convolutional layers must perform mathematical operations on large amounts of high dimensional RGB images. Many researchers have recommended to use transfer learning, which essentially reuses an architecture that was pre-trained on a different dataset. Sanni, et al. (2024) designed an automated tea leaf clone classifier using a VGG16 based custom CNN in an attempt to make a lighter model. They extracted features using the convolutional layers of VGG16 and built a custom model on top by replacing VGG16's original dense layers with batch normalization and dropout layer. By conducting hyperparameter tuning with different batch sizes, dropout, and learning rates (LR), they optimized the model. The model achieved the highest validation accuracy of 93.24% using the smallest image batch with low dropout and learning rates. The model generalized well on unseen data, with only 0.02% less test accuracy than the validation. Confusion matrix suggested that each class had reliable accuracy. While the previous research mainly focused on working around a single classifier, Shijie, et al. (2017) proposed to compare two different models for detecting tomato pests and leaf diseases. In the first method, they extracted image features from VGG16's second dense layer and trained them with a traditional machine learning classifier SVM. Alternatively, they fine-tuned the same network by replacing two fully connected (FC) layers with one layer containing half the neurons, making the final output dense layer the classifier. They uncovered that the fine-tuned model performed with 89% accuracy, which was slightly better than the one with SVM which was 84%. However the higher accuracy of the fine-tuned model came at the cost of higher training time.

A limiting factor in the last two studies is that they focused on smaller and balanced datasets to achieve fair performance across all classes. In contrast, the proposed research attempted to use larger and imbalanced dataset to acknowledge the difficulty in identifying underrepresented classes, which is essential for situations like rare diseases with lack of data. This can help to make a robust and flexible model that functions better in real-world scenarios. Furthermore, the discussed studies do not rationalize the choice of the hyperparameters and search spaces. The proposal differs, because it employed a random search optimization method to locate a reasonable starting point, then gradually explored different combinations around it. While Banerjee, et al. (2024) fed the estimations of the Random Forest classifier to the output layer of CNN for final prediction, this proposal attempted to use Random Forest as one of the machine learning classifiers, similar to the approach by Shijie, et al. (2017). However, instead of using the learned features from the fully connected layer, the proposed research followed the feature extraction method of Sanni, et al. (2024) by only using the convolutional layers, disregarding the FC layers. A key difference in the proposed method is that instead of using a custom model as Sanni, et al. (2024), it attempted an integrated approach by using VGG16's convolutional layers as the feature extractor. Then, the features were trained using both Random Forest, and a customised CNN architecture to compare their performance.

This paper is structured into nine sections. Followed by the introduction, Section 2 outlines the main objectives in detail. Section 3 discusses related works about ML and CNN in the field of agriculture. The

main methods to design the project is discussed in Section 4, which covers details about the dataset and its distribution, feature extraction from the preprocessed data using transfer learning, dimensionality reduction for ML, class balancing methods and finally discuss about the ML and CNN architectures. This section ends the evaluation techniques as well as the technical aspects for the entire procedure. In section 5, the training process of the models are explained and rationalized in detail. The description of their functionalities include hyperparameter tuning, optimization techniques and experimentations.The experimental results are analysed in Section 6. With the best performing models identified from ML and CNN, their evaluation matrices are discussed in Section 7, which helped to identify the final image classifier. Section 8 discusses web application development and the research is concluded with Section 9.

2 Aims and Objectives

The aim of this research is to design two Tomato leaf disease classifiers, this will be done by integrating a traditional Machine Learning model and Convolutional Neural Network architecture to a DL based feature extractor and compare their classification performances to select the better one for web deployment. To achieve the objectives, several existing literatures will be reviewed to explore potential ML models, CNN architectures and feature extracting methods. The literature surveying will help to analyse the strengths and limitations of existing methods to effectively design and improve the proposed research. The main objectives are to:

Data Preparation: Choose a relevant tomato leaf disease dataset for the learning process, ensuring it represents real-world plant disease aspects.

- ✓ Prepare the dataset by cleaning irrelevant aspects.
- ✓ Divide the original dataset into separate portions for training and model evaluation.
- ✓ Pre-process the data by resizing, label extracting, and image augmenting.
- ✓ Check for class imbalance in the dataset to prevent classification biasness and identify methods to address the disparity.

Feature Extraction: Investigate different DL architectures to identify effective ways of utilizing the proposed network for comprehensive feature extraction.

- ✓ Explore the usefulness of transfer learning in predictive analysis.
- ✓ Choose the feature extractor by considering its performance on the benchmark and other relevant models.
- ✓ Extract the features and decide methods to build a suitable architecture on top of the feature extractor.

Model selection and Preparation: Based on the original proposal (Anower, 2024),

- ✓ Identify the ML and CNN models that are to be used.
- ✓ Compare the proposed ML models to discuss why the selected one is more useful for the type of problem.
- ✓ Rationalize the decision of designing a custom CNN architecture instead of leveraging existing architectures from the proposal or fine-tuning the proposed feature extractor.
- ✓ Identify necessary methods to make the extracted 2-D features compatible for the classifiers to perform mathematical operations on.
- ✓ Reduce the high dimensional features into low dimensional principal components before training the ML model.

Hyperparameter tuning and Model Training: Define search spaces to perform hyperparameter tuning for the models.

- ✓ Train the ML model using default parameters of the classifier.
- ✓ Optimize both proposed models using Random Search to determine the optimum hyperparameter combination from the search space.
- ✓ Justify the parameters used and the limit of the search domain.
- ✓ Use the best hyperparameter combinations to train the models.
- ✓ Validate the models on unseen data.

Experimentations: Use the optimum hyperparameter as a reference point to:

- ✓ Justify the choice of the hyperparameters to conduct the experiments with.
- ✓ Adjust the hyperparameters with higher and/or lower values to train, validate, and test the models.
- ✓ Analyse the effects on feature learning and generalization performance.
- ✓ Use the experimental results to decide the better model from both classification methods.

Model Evaluation: Use the results and evaluation techniques to compare the classifiers.

- ✓ Use Performance metrics of both classifiers to assess the accuracy in classifying each category.
- ✓ Discuss overall performance, along with strengths and weaknesses due to architectural characteristics.
- ✓ Based on the evaluations, finalize the better classifier.
- ✓ Compare the final classifier against benchmark models that used the same dataset and discuss why it performed better or worse than the benchmarks.

Web Deployment: Create a user interactive webpage.

- ✓ Deploy the final classifier into the Backend.
- ✓ Test the classification by uploading images that are unknown to the trained model.

3 Literature Reviews

Although the initial exploration of artificial intelligence in agriculture commenced by the end of the 20th century, the rapid development did not start till the 2010s. That is when CNN achieved breakthroughs in image classification using the ImageNet dataset. As computing resources got powerful with large amounts of data being widely available, automated systems emerged dramatically over the last few decades. This, along with computer vision has helped scientists to advance their research in precision agriculture and plant pathology. The following section will discuss several studies involved in classifying tomato leaves as well as some other essential crops.

Islam, et al. (2023) reviewed VGG16, VGG19 and ResNet50 architectures to create a smart web application called DeepCrop. It could analyse plant photos in real time to distinguish between infected and healthy leaves. Multiple experiments with different batch sizes from a small dataset concluded that ResNet50 performed with better accuracy of 98.98%, and significantly lower testing errors using higher batches, and dealt with vanishing gradient problems, unlike the other two architectures. To reduce the long convergence time of a CNN model to classify corn leaves, Hu, et al. (2020) applied transfer learning to improve a GoogleNet architecture by adjusting the parameters in its dense layers. To train the model, they used a small dataset and dynamically adjusted the training hyperparameters to determine the optimal setting that can avoid overfitting issues. Upon comparing it with the original GoogleNet, the improved model portrayed higher accuracy of 97.60% and a lower loss value with a short training time. A CNN based edge computing system was proposed by Durmuş, et al. (2017) to spot tomato leaf diseases and presence of harmful pests. Their study focused on examining the performance of different sized architectures for a moderate to large dataset. The model evaluation showed that the bigger model AlexNet had slightly higher test accuracy. However, they selected SqueezeNet for the platform due to its lightweightness but competitive results, given that it was 80 times smaller. This work highlighted the possibilities of lightweight CNN architectures for mobile computations in agricultural applications. Prasvita & Herdiyeni (2013) created an android based mobile application called MedLeaf to identify medicinal plants using image processing and RBF based Probabilistic Neural Network (PNN). They used Local Binary Pattern Variance (LBPV) to extract localized patterns of different texture and contrasts from the leaves. Their classification accuracy was only 56.33% because LBPV did not capture enough features from image data , as the dataset was inadequate in quality.

While the previous studies focused on Neural Networks, researchers have also been working on utilizing traditional machine learning models to develop and improve plant disease detection systems. Machine learning is often recognized for better interpretability and performance on limited resources. To resolve the agricultural challenges faced by farmers, Prabavathy, et al. (2023) investigated five different classification models such as SVM, K-Nearest Neighbours, Decision Trees, Random Forest, and a CNN to identify the least complex classification system with higher accuracies. The relevant features like Mean Green Intensity were extracted from a moderate to large sized dataset of multi-leaf images using a pretrained VGG16. The results suggested that in terms of processing time and accuracy results, Random Forest outperformed the rest with 92% perfection, proving the effectiveness of merging traditional ML with transfer learning for plant disease detection. To classify a very small dataset of paddy leaves using ML models, M, et al. (2017) extracted diseased attributes by segmenting the infected regions from RGB images using Hue, Saturation, Value (HSV) colour model. Then they extracted properties of the segmented regions using geometrical measurements like the area, axis, and perimeters. Using k-Nearest Neighbour, the leaf categories were predicted with 76.59% accuracy. Observations made from reviewing these researches is that the classification performance is often based on the size of the dataset, the method of feature extraction and the magnitude of the network's architecture.

The application of machine learning has received significant interest in the direction of plant disease recognition due to its success rate and exactitude. Researchers have constantly recommended diverse approaches to overcome the existing challenges. However in the field of agricultural diagnostics, challenges still persist due to the lack of data on rare diseases, or only due to the availability of lab-captured images that may not represent the unique diseased characteristics depicted by actual vegetation in the fields.

4 Methodology and Methods

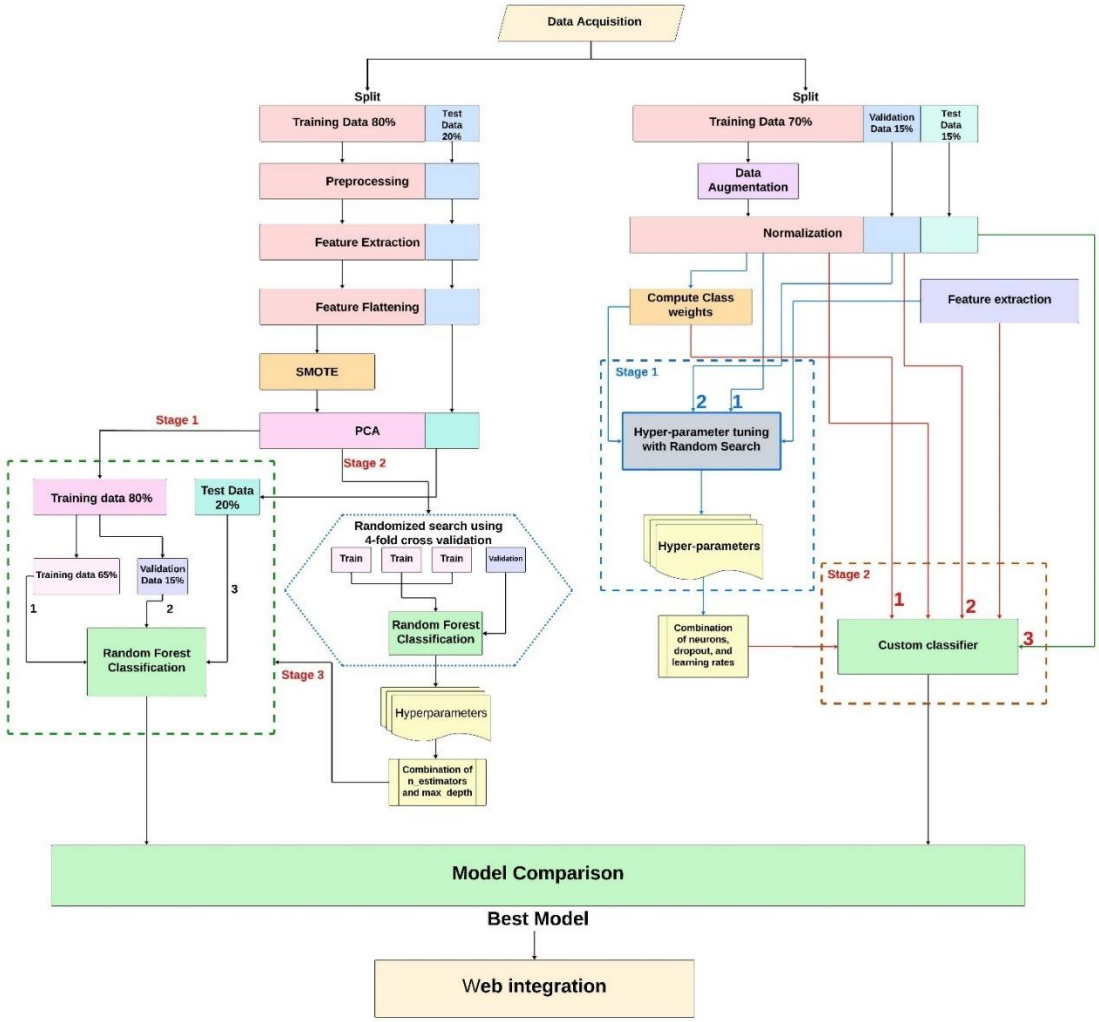


Figure 1: Project Workflow

Figure 1 shows the workflow of the proposed research. The implementation methods and design specifications described in the workflow will be discussed in Section 4 – 8.

4.1 Data Acquisition

As this project focuses on classifying diseases related to tomato plants, a tomato leaf dataset was used from Kaggle. "PlantVillage" was retrieved from Kaggle (Emmanuel, 2019), which is a diverse dataset containing about 342.23 MB of image data, comprising 20.6k files. The image library originated from the PlantVillage project researched by Hughes & Salathé (2015) in Penn State University.

Figure 2: Randomly generated phenotypes of tomato plants from each class directory of training data

The Dataset formerly accommodated 15 folders, representing 15 different classes. Each class acted as a healthy, or a type of disease present in various plants, such as Peppers, Potatos, and Tomatoes. There were 5 classes for Peppers and Potatoes combined, and the remaining 10 belonged to tomato leaves. The proposed research only focused on tomato leaf diseases, hence only 10 folders dedicated to the tomato leaves were used. It included folders for 9 diseased categories and 1 healthy category. The original dataset also contained duplicate folders; so, after removing all the irrelevant ones, it finally resulted in 247 MB of data with 16,012 images in total. The categorical labels were converted to numerical labels for mathematical operations. The values were assigned in the ascending order of the alphabet, as shown in Table 1.

The overall data distribution in each class was imbalanced with a very low proportion of data in some classes like *Mosaic virus*, while extremely high on *Yellow Leaf Curl Virus*. This could result in biased prediction towards the majority classes. Techniques to address this imbalance are discussed in Section 4.3.

index	Class (exact labels)	Number of Images	Percentage
0	Tomato_Bacterial_spot	2127	13.28%
1	Tomato_Early_blight	1000	6.25%
2	Tomato_Late_blight	1909	11.92%
3	Tomato_Leaf_Mold	952	5.95%
4	Tomato_Septoria_leaf_spot	1771	11.06%
5	Tomato_Spider_mites_Two_spotted_spider_mite	1676	10.47%
6	Tomato__Target_Spot	1404	8.77%
7	Tomato__Tomato_YellowLeaf__Curl_Virus	3209	20.04%
8	Tomato__Tomato_mosaic_virus	373	2.33%
9	Tomato_healthy	1591	9.94%
	Total	**16012**	**100%**

Table 1: Class names and number of images in each class

4.2 Data Division

The loaded dataset was initially divided into two parts for ML, and three parts for CNN using *scikit-learn*'s `train_test_split()`. The training set was used as input data to go through learning processes by adjusting various model parameters. A validation set was preserved for the CNN model and was used to check the model's generalization ability on unseen data during the training process. For the ML, the validation portion was extracted from the training set right before model training, so it contained similar characteristics as the training data. The remaining data was used as a test set. This portion was completely unknown to the trained model and responsible for determining the classifier's effectiveness in real time.

Once the best models are determined with the help of hyperparameter optimization techniques, testing these models helps to give an unbiased estimate of the performance quality on unseen data. After splitting the dataset before preprocessing, they were saved to designated folders. The test data folder was downloaded immediately after splitting, to use for web testing.

The general rule is to have a high model training accuracy score, as well as a high accuracy score when evaluated with new and unseen data. This balanced phenomenon proves that the generalization ability of the classifier is excellent. However, if the gap between training and testing accuracy is too high, a higher training accuracy score may be a sign of overfitting (Xu & Goodacre, 2018), and vice versa.

The initial proposal was to split the entire dataset into three parts after pre-processing: training=70%, with validation and testing each 15% (Anower, 2024). However, during the actual implementation, the dataset was split first and pre-processed only after. For the ML model, the current proposal divided the training and test data in 80:20 ratio, with an additional validation split of 15% kept aside from the 80% of training data before training. So, there were 12,683 training samples and 3,174 test samples for ML. For CNN, the data splitting was kept the same as the initial proposal with 11,201 samples used for training, 2,405 for validation and 2,406 for testing, in a 70:15:15 ratio.

4.3 Data Preprocessing

For the ML model, the training and validation data were pre-processed separately after splitting. Each image data was loaded in colour mode from their directories using *OpenCV*. To maintain consistent dimension, all the data were resized to *224 × 224* pixels square images. Each image belonged to a class with a categorical label and were encoded to integer values based on alphabetical order using `preprocessing.LabelEncoder()` from *scikit-learn library*. The pixels were scaled to values between 0 and 1 to keep the proportions identical across the dataset (Banerjee, et al., 2024). This normalisation of pixel intensities is a general rule before passing image data to any Convolutional Neural Network for extracting features, because ML models do not perform well with inputs of different numerical scales (Géron, 2019). The input data were originally 8-bit images with variable pixels between 0 and 255. By simply dividing the data sample by 255 as in equation (1), the pixel values were shifted to a range between 0 and 1.

$$\mathbf{X\_norm} = \frac{\mathbf{X}}{\mathbf{255}} \qquad \mathbf{(1)}$$

where X is the original data sample such as training or test samples.

To artificially diversify the fixed sized data and prevent the proposed CNN model from retaining too specific patterns, data samples were modified using augmentation with the help of `ImageDataGenerator()` library from `Keras`. For the training data, translation techniques like height and width shifting were added for positional variations. The images were also skewed along the x-axis to introduce horizontal distortions, and zooming was applied to imitate deviation in image size like real world scenarios. While augmentations were only applied to the training data, all the data were rescaled to normalize pixel values between 0 and 1 for consistency. These adjustments enable the CNN model to perform well on unseen data and dependably categorise images in real-time (Banerjee, et al., 2024). Data Augmentation helps to build a robust model by simulating variations and noise such as different point of views, lighting, obstructions etc, and also helps to train the under-sampled classes effectively (ccslearningacademy, 2024).

To create a data pipeline for training and evaluation of the CNN using images, a `.flow_from_directory()` method was used that can load image data from their directories automatically and label them based on the receding subdirectories. Each image was loaded in *224 × 224* pixels in batches of 128 to optimize memory usage. If the batch size is too high, even if the training process becomes faster, it can use up computational resources. With a small to moderate batch size of 128, the model can learn by making finer adjustments for longer periods and produce more reliable accuracies (Sanni, et al., 2024). The classes were one-hot encoded using *class_mode='categorical'* to vectorize the target labels with 1s, and others as 0s. For example, in the proposed dataset, there are 10 classes, and if the prediction points towards class 3, the encoded form will take the shape of [0010000000] (Aggarwal, et al., 2021). The categorical labels in the generators were assigned in alphabetical order using *shuffle=False*.

4.4 Feature Extraction

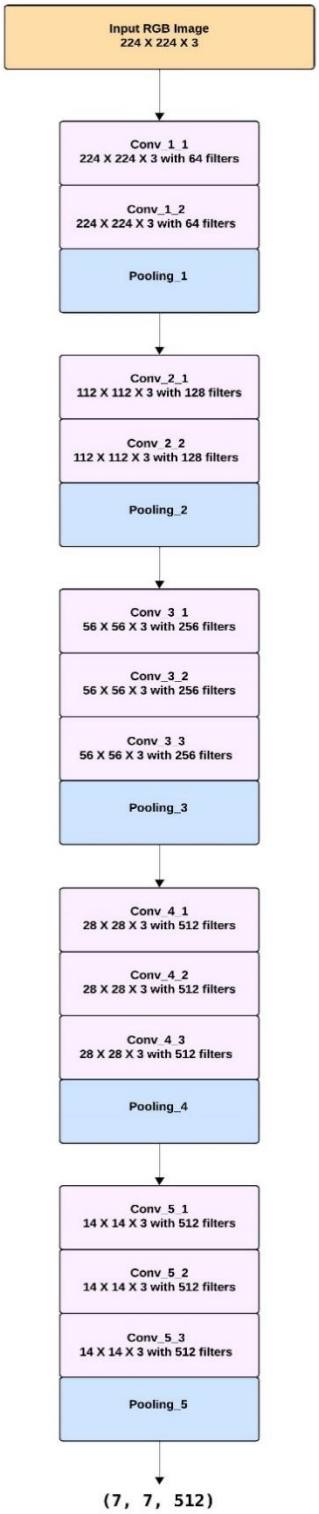


Figure 3: Visual representation of VGG16 architecture for extracting feature maps

To classify plant diseases using the proposed ML and CNN architectures, first the relevant features were needed to be extracted from the pre-processed data. The proposed method for feature extraction was to utilize DL frameworks like TensorFlow and Keras to load a pre-trained CovNet architecture called Visual Geometry Group 16-layer network, AKA VGG16. It is an improved version of AlexNet (Chakraborty, et al., 2024), and was originally trained by ImageNet, a large dataset that contained 1.26 million images from 1,000 classes (Shijie, et al., 2017). The model was trained using images of fixed size of *224 × 224* with RGB channels, thus the input image must also be of a (224, 224, 3) tensor to accomplish transfer learning (pawangfg, 2024). This network can extract a large number of features from the PlantVillage dataset by utilizing the learned representations and patterns from ImageNet. The technique is known as *inductive transfer learning*, which remembers the weights from certain observations and uses them to extract features from a different dataset (Pardede, et al., 2021). To train a CNN architecture from scratch requires a large amount of data and is very time consuming. That is why, transfer learning can be a great alternative to extract low-level detailed features from different convolutional levels (Hu, et al., 2020). The features will be combined gradually at each convolutional layer into complex representations.

Although VGG16 is a 16 layered model, only the 5 convolutional blocks of 13 layers were used for feature extractions, and the top 3 dense layers (2 FC layers and an output FC layer) were omitted by setting VGG16's *include_top=False* (Sanni, et al., 2024), as shown in Appendix A. Figure 3 shows that each convolutional block has sets of convolutional layers with a pooling layer at the end. The convolutional layers apply filters that slide across the input feature map to calculate the dot products between weights and pixels, extracting features such as edges and textures. Each convolutional layer has a ReLU activation function after it to introduce non-linearity, enabling the network to learn further detailed patterns (Géron, 2019). The pooling layer reduces the spatial size of the extracted features from the previous layer but retains all its substantial information (Sanni, et al., 2024).

The spatial dimensions of the output image must always remain constant as the input; thus, the padding is always kept "same" or 1 (Géron, 2019). To determine how much filter moves across the image data, a stride value of 1 pixel with a 3 X 3 window is used for the convolutional layers. However, the pooling layers use 2 pixels of stride with a receptive area of 2 X 2 window size. Doubling the stride halves the activation size to reduce the spatial dimension.

A similar process repeats for all the 5 blocks, where each hidden layer imposes different filters to extract distinctive features. When a uniformly resized image was fed through the input layer, it passed through the first block of 2 convolutional layers, both containing 64 filters. At the pooling layer, the spatial dimensionality was reduced from [224, 224, 64] to [112, 112, 64].

Similarly, as the data passed up each block, the dimensions were halved, and the filters were doubled. By the end of the last block, the dimension of the feature map was reduced to [7, 7, 512], representing a height of 7, width of 7, and depth 512, implying 512 different channels (Sharma, et al., 2022). The structure of these layers aids the model to acquire intricate hierarchical representations of visual features (pawangfg, 2024). Appendix B illustrates the architecture and its parameters. For this research, all the layers in the feature extractor were configured to be non-trainable by freezing the pre-trained weights. Thus, the abstract features from ImageNet were hierarchically transferred to the tomato leaf dataset without being altered during training.

At the end of block 5, the features maps that have been gathered by the convolutional and pooling layers were generated in 2-D raw forms, which needed to be restructured to a 1-D flat vector before transferring to the classifiers. To flatten the feature maps for the ML model, a `reshape()` method was used, where NumPy automatically calculated the size of the features and turned them into a single vector 25,088. For CNN, the flatten layer converted the 2-D feature maps into the 1-D vector using `Flatten()` method, to extract high-level features that are not interpretable in their raw form**.** Flattening can be mathematically represented as equation (2).

$$\mathbf{X} = [\mathbf{x1}, \mathbf{x2}, \mathbf{x3}, \ldots\ldots, \mathbf{x25088}] \quad \textbf{(2)}$$

4.5 Class Balancing

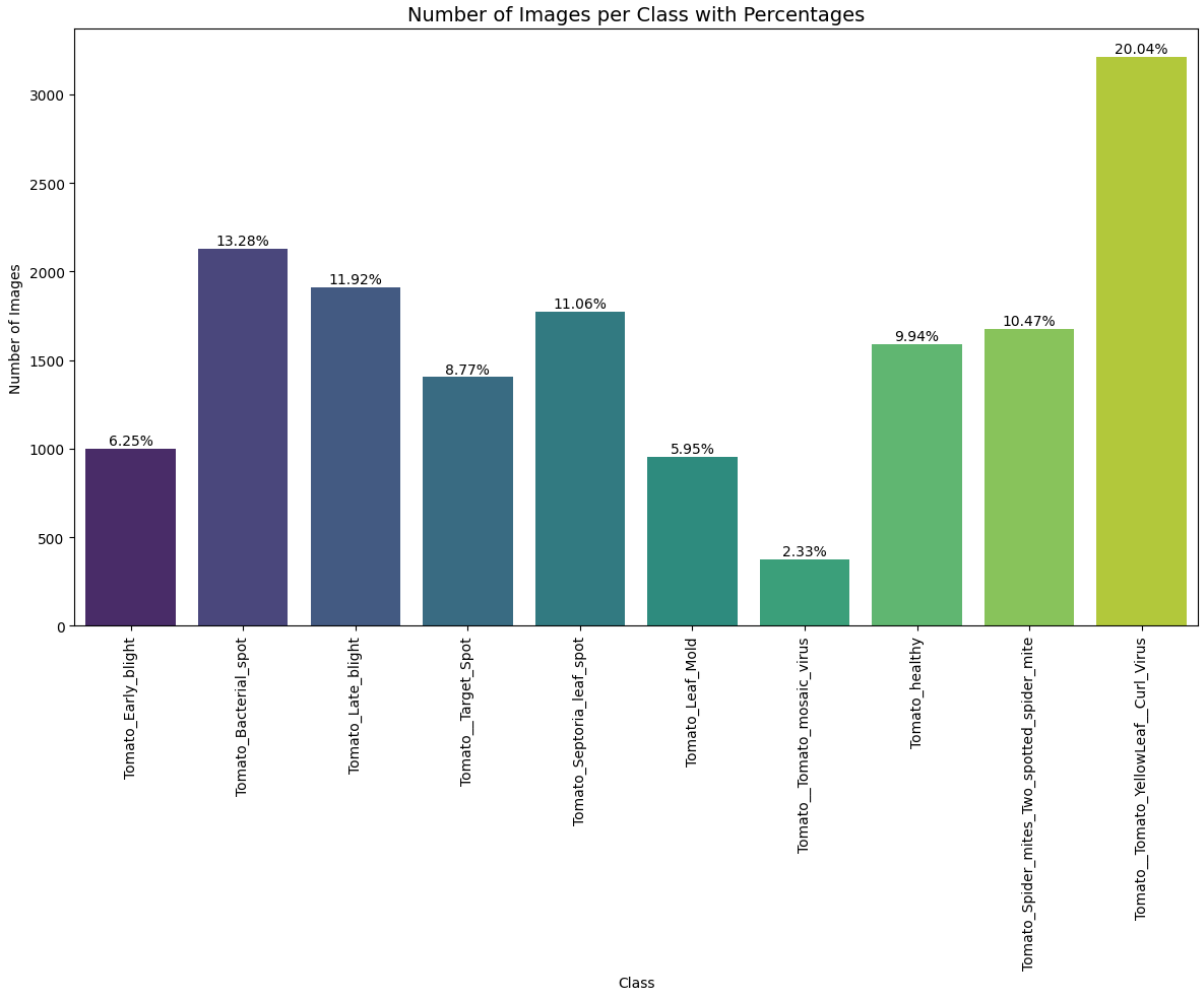


Figure 4: Class Imbalance

Table 1 shows that the data distribution across the classes in the original dataset was noticeably imbalanced. While most classes had moderate distribution between 8.8% to 13%, some categories such as *early blight*, *mosaic virus* and *leaf mold* were understated, with as low as 2.3% of the total data. Contrarily, the *yellow leaf curl virus* group was very dominant with 20% of the total data, as shown in Figure 4. This imbalance could degrade the training process by making trained models biased towards the overrepresented classes and not learn enough meaningful patterns from the minority classes (Ahmad et al., 2021). These types of models can often achieve high overall accuracies by only classifying majority classes correctly but underestimate the probabilities of the rare events with low recall values (He & Cheng, 2021).

For the ML model, the training data was balanced using Synthetic Minority Oversampling Technique, which essentially over-sampled the minority classes by generating synthetic samples. It is done by taking each sample from a minority class and creating synthetic points along the line segments that lie between the sample and its nearest neighbours (Chawla, et al., 2002). These additional samples were generated automatically by augmentation techniques like rotation and skewing. With the `SMOTE()` class from the *imbalanced-learn* library, the minority classes were over-sampled using *sampling_strategy='auto'* parameter to match the prevalent class's sampling size. Method `fit_resample()` applied SMOTE to

the training data only, making 2,566 samples in each category. This resulted in a total 25,660 samples of training set. Note that the validation set was extracted from this augmented training data before training the models, as described in Section 4.2.

For CNN classifier, the imbalanced training dataset was handled using Inverse of number of samples (INS), where the weights of each class were calculated as the inverse of the class frequency times the total number of samples in the dataset (Bakirarar & Elhan, 2023). For example, in Early Blight, there were 1,000 data in total, with 699 training data after separating 70% (11201 total). Using INS equation (3), the weight was calculated as W = 16.02. This inverse relationship authorized the model to prioritize the minority classes with larger weights and popular classes with smaller weights based on the distribution in each class. This helped to alleviate the misclassification of the rare events.

$$\textbf{weight of class i} = \textbf{total training sample} \times \frac{1}{\textbf{number of training samples in class i}} \quad \textbf{(3)}$$

Using *Scikit-learn*, the class weights were balanced automatically by setting parameter *class_weight='balanced'* in `compute_class_weight()`. The class weights calculation in equation (4) used the total number of classes in the dataset to scale down the weights, making the class weight of Early Blight W = 1.60. In equation (4), *np.bincount(y)* counted the number of samples in a given class; for example, 699 samples in Early Blight's training set.

$$\textbf{class\_weight} = \frac{\textbf{n\_samples}}{(\textbf{n\_classes} * \textbf{np.bincount}(\textbf{y}))} \quad \textbf{(4)}$$

These calculated weights were introduced during the training process to adjust the loss functions by giving higher priorities to the classes with higher weights.

4.6 Dimensionality Reduction for Machine Learning

After feature extraction with transfer learning, the resulting feature dimension was 25,088, which was considered very high. Not every feature might be informative to the target variables. Random Forest classifiers do not have specific criteria to select appropriate features, but to sample at random (Fiagbe, 2021). Additionally, with 25,660 samples having as many dimensions as the sample size, the computational time and complexity would be very high with low predictive performance in the ensemble learning. This phenomenon is known as the "Curse of Dimensionality" which can downgrade the efficiency of the classification performance by learning irrelevant feature patterns. Therefore, it is important to reduce the dimensionality to project the training data into a lower dimensional hyperplane that lies closest to the actual data (Géron, 2019).

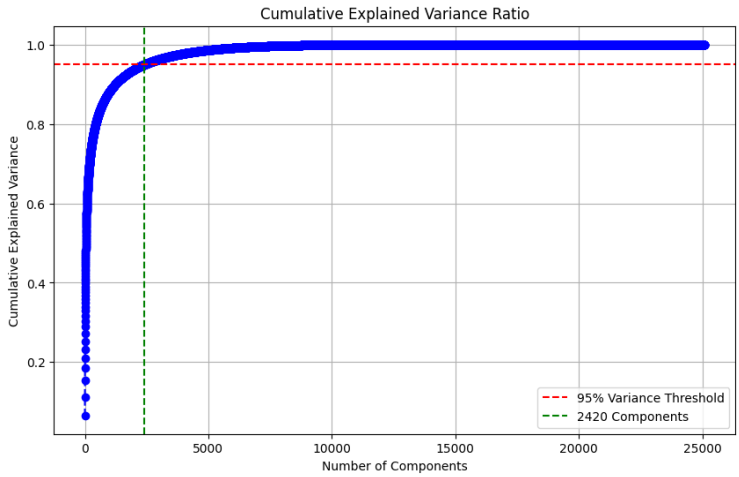


Figure 5: PCA components for 95% variance

Principal Component (PCA) is a dimensionality reduction technique that reduces the distance between the data distribution of different sets of data in the latent space (Bharadiya , 2023). Not only does this improve training time, but also eliminates data that are noisy, redundant, and irrelevant for classification. PCA performs further feature extraction to transform the original features into principal components by lowering the complexity but retaining most of the data information (Velliangiri, et al., 2019). While reducing the dimensionality in this research, at least 95% of the training data information was retained (Géron, 2019). Figure 5 describes that to preserve 95% variance of the training sample, 25,008 features can be transformed into 2,420 principal components. From this, it is advocated that the dimensionality could be reduced up to 2,420 features and still be truly informative. This approach was taken to improve the resource efficiency and classification accuracy of the proposed Random Forest model. The test data was projected onto the same principal components that were acquired from the training data.

The reduced features were visualised in 2D using t-SNE to understand the complex feature relationships in the dataset. Appendix C shows that several classes had overlapping data points, suggesting that these classes had similar features. t-SNE essentially further transformed the 2,420 principal component into 2 components.

4.7 Models and Architectures

4.7.1 Machine Learning Algorithm

The original proposal was to select one of the ML models between SVM and RF, and the latter is finalized because of its several advantages. While both models are supervised learning algorithms, SVM isolates the dataset into two classes using a hyperplane whereas RF combines the predictions of several decision trees to forecast among multi-class targets. SVM is an effective image classifying model with accuracy in high dimensional space but falls short when it comes to the size of data. With a medium to large dataset of classes with overlapping features, as shown in Appendix C, the training time of SVM can be very high. To overcome that challenge, a non-linear machine learning algorithm like RF can be utilized as categorical data modelling to identify the non-linear patterns in high dimensional image data (Ali, et al., 2012; Titapiccolo, et al., 2013).

RF can efficiently deal with high dimensional large datasets by training multiple decision trees in parallel (Samarth, 2023). RF uses ensemble procedures such as bootstrap aggregation to build each decision tree. The technique involves taking random samples from the entire training distribution to build a tree in each forest (Breiman, 2001). To construct the tree, an in-bag training subset is sampled, with a smaller out-of-bag test portion to evaluate the resulting tree's performance. Although each tree chooses subsets randomly from the same population, some subsets may end up with repeated data points, while some data-points may not be included at all. Once each decision tree in the forest makes unit prediction on the most popular class for a predictor x, a majority voting system evaluates all the votes from the ensembled trees into a single output to classify the same predictor (Rigatti, 2017; Banerjee, et al., 2024). RF is a good choice for classifying multi-class disease detection because the algorithm does not overfit even if more trees are added to the forest, thus not requiring pruning too (Ali, et al., 2012). By overcoming overfit issues, the classification becomes more robust to noise, with better generalization ability in real-time. RF can create a large number of decision trees for a large number of instances (Rigatti, 2017); however, after a certain point, more trees do not improve the generalization ability anymore, so the error reduction stabilizes (Breiman, 2001). Additionally, the computational intensity also increases with increasing trees. While building the trees in the ensemble, the maximum number of features picked up by each node is set to "sqrt", so it can pick square root of the total number of features at a node (Rigatti, 2017). For example, PCA obtained 2,420 features in Section 4.6, thus there can be maximum $\sqrt{2420}$ = 49 random features at each node during a split.

The training time can be improved considerably by assigning a minimum value for node splitting, so a splitting will occur only if a node contains that number of samples. However, if the minimum sample for splitting is too low, the trees can grow deeper and capture more complex patterns. The tree depth decides the complexity of the Random Forest and gets more compound as the trees grow deeper (Al-Janabi & Andras, 2017). However, the accuracies often stop improving after a certain depth (Nadi & Moradi, 2019). Setting a minimum number of leaves for each node helps to control the growth of trees. Higher number of minimum leaves causes a node to split less often, which can lead to a simpler model with low performance (Al-Janabi & Andras, 2017). During a node splitting, a time efficient quality measure called Gini impurity is calculated by evaluating all the splits in that node and finalising the split with lowest impurity score (Géron, 2019).

4.7.2 CNN Architecture

The initial proposal was to use one of the DL architectures from AlexNet, VGG16, or a customised model. The final decision is to build a customised model on top of the pretrained VGG16 feature extractor.

Several factors were considered before choosing the DL model to work with. For example, AlexNet has eight less convolution layers compared to 13 layered Vgg16, whereas contains larger (11 × 11, 7 × 7, 5 × 5) convolutional filters compared to VGG16 with multiple stacked (3 × 3) filters. Having a higher number of convolutional layers and smaller convolutional cores gives VGG16 the advantage to create a deeper network with fewer network parameters, thus it can capture fine-grained features and improve performance efficiency (Yang, et al., 2021). Because of its deeper architecture, the 13 convolutional layers of VGG16 were used as the feature extractor for the DL model, as mentioned in Section 4.4. However, to build a complete classifier, it was considered to add a customised FC layer after flattening the feature maps from VGG16 instead of using the existing full architecture. This is because the top three layers of VGG16 learn class-specific features, and since it was originally trained on ImageNet with categories that are unrelated to the proposed dataset, it can lead to learning irrelevant patterns that are not generalizable; thus, result in overfitting.

To make a well interpretable model with a smaller no. of parameters, only one FC layer containing fewer dense units was used to add additional non-linearity for learning complex patterns in the data. The custom model was tuned with a range of neurons, dropout layers, and a fixed regularization and optimized with different learning rates. Depending on the number of units in the FC layer, a model can learn specific patterns from the high-level flattened features using ReLU activation function. For example, if there are N neurons, each receives all features from the flatten layer, multiplied by randomly assigned weights. The process can be described using equation (5):

$$\mathbf{Z_j} = \sum_{\mathbf{i=1}}^{\mathbf{25088}} (\mathbf{w_{i.j}} \,.\, \mathbf{x_i}) + \mathbf{b_j} \qquad \textbf{(5)}$$

where, $w_{i.j}$ is the corresponding weight for i^{th} input feature and j^{th} neuron, x_i is the i^{th} input feature from the flatten layer (equation 2), and b_j is the bias term for the j^{th} neuron, which helps adjust the output of the neuron.

With a biased term added to the weighted sum of all the inputs from the previous layer, the result was passed to the activation function to generate the final output of each neuron. With too many neurons, the model may learn too well, while less neurons can underfit the model. The role of the activation function (equation (6)) is to learn non-linear complex pattern by making negative Z_j values to 0 or allowing positive Z_j as output.

$$\mathbf{ReLU}(\mathbf{Z_j}) = \mathbf{max}(\mathbf{0}, \mathbf{Z_j}) \qquad \textbf{(6)}$$

This reduces the chances of vanishing gradient problems - a phenomenon where some neurons are unable to update their weights effectively during training as the network deepens, and the gradient becomes very small (Kiliçarslan & Celik, 2021). However, while preventing the dying of the weights, it may activate too large of weights and learn the patterns too well, causing the model to memorise them. To further prevent the model from overfitting, L2 weight regularization technique is applied, which adds a regularization term 0.01 to the loss function and constraints large weights by summing the squares of the weights (Brownlee, 2020; Géron, 2019)

Followed by the FC layer, a dropout layer was introduced to randomly turn off some neurons based on the dropout rate during each training iteration to ensure that certain neurons do not dominate within the network. For example, if a dense layer has 1,024 units and the dropout rate is 0.25 (25%), 256 random neurons will be turned off and the remaining 768 will actively learn the training data. This mechanism helps to prevent overfitting, because it enables the model to learn from a unique combination of neurons during training. The output layer contained the same number of dense units as the proposed no. of classes, with a SoftMax activation function. The same process as a FC layer took place in this layer to produce probability values for each class, with the highest probability class as the prediction result for an input image (Sanni, et al., 2024). Adaptive Moment Estimation algorithm ADAM was used to improve the accuracy of the network. This optimizer essentially uses gradient first-order and second-order moment estimations to adjust the learning rate of the weights and biases of the neural units during back propagation (Hu, et al., 2020).

4.8 Evaluation Metrics

Once the best ML and CNN models were decided based on their high overall accuracy and small generalization gaps (in Section 6), each model was evaluated using its *confusion matrix* and *classification reports*. The confusion matrix showed how many instances the classes of a model predicted accurately, while the classification report outlined the *precision, accuracy, recall* and *F1-score* of each class with the overall weighted average scores of the models.

Keywords: *True positive* ***(TP)****, True Negative* ***(TN)****, False Positive* ***(FP)****, False Negative* ***(FN)***

The *accuracy* tells the amount of test cases that are predicted correctly, while *precision* is the ratio of correctly predicted instances to all positive instances.

$$\text{Accuracy} = \frac{TP+TN}{TP+FP+FN+TN}$$

$$\text{Precision} = \frac{TP}{TP+FP}$$

The percentage of all the correct predictions relative to the total positive instances is known as the *recall* (Islam, et al., 2023).

$$\text{Recall} = \frac{TP}{TP+FN}$$

Lastly, *F1* score is the harmonic mean of the combined precision and recall (Kundu, 2022).

$$\text{F1 score} = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision}+\text{Recall}}$$

The confusion matrix shows the TP values diagonally from top left to bottom right, that is when the model predicts an instance as an infected leaf when the leaf is genuinely infected. The *recall* in the classification report gives the % result of the TP of each class against the total support of that class. The weighted average scores give the overall result of the classifier.

4.9 Hardware and Software Requirements

To take advantage of the cloud computing platform, the proposed tasks such as data preprocessing, feature extractions, and model training were conducted in Google Collaboratory. It is a cloud-hosted version of Jupyter Notebooks that allows users to utilize its high-end computing resources like GPU, TPU and larger Discs. The dataset that was saved in Google Drive was accessed directly from Colab's environment. However, a challenge was that every time the session terminated, the data needed to be reloaded into the environment. Therefore, the testing set was manually downloaded from Colab into the local environment immediately after splitting so it could be used for testing the classifying web application later. At the end of each experiment, the best model from each classifier was also downloaded into the local environment to deploy in the backend. The programming was primarily completed using Python, with ML and DL frameworks.

The webpage was constructed in Visual Studio code using backend frameworks like Flask. The frontend was designed using HTML and CSS. JavaScript worked as a bridge between the backend and the frontend, which is discussed further in Section 8. To store the details of Tomato leaf diseases, NoSQL server was used. See Appendix D for a detailed list of all the software and hardware used in this research.

5 Implementations and Training Process

5.1 Random Forest

5.1.1 Base model

Stage 1 in Figure 1 shows that after splitting 15% of validation samples from the original training data, the RF model was trained using `RandomForestClassifier()` class from *scikit-learn*'s *ensemble* module (see Appendix E). The class was initiated using its list of default parameters such as 100 trees, with maximum depth allowed to grow till all leaf nodes have less than 2 samples. Table 2 describes the full list of default parameter values offered by the classifier.

5.1.2 Hyperparameter Tuning with Random Search

While the previous method explored the default parameters set by the algorithm, an attempt was made to improve the model's performance by using Random Search with Cross Validation in Stage 2, as shown in Figure 1. *Scikit-learn* offers classes like `RandomizedSearchCV` to locate the model with best hyper-parameter values from a given search space that can produce efficient predictive performance (Mantovani, et al., 2015). To optimize the classifier's performance, first a search space was defined using a restricted range of RF hyperparameter values, so that the searching method could pick a combination of parameters from the space randomly. The search range was not kept too broad to minimize computational time. For hyperparameter tuning, the *n_estimators* were set from default value 100 to 500, so the algorithm could identify the best number of trees in less time. The upper bound was kept till 500, as increasing trees is computationally intensive (Rigatti, 2017), with additional search time. As shown in Table 2, the *max_depth*, *min_sample_split*, and *min_sample_leaf* were provided with low values to capture finer details, and increased gradually to a maximum limit so the algorithm could make diverse combinations to learn different levels of complexities. With bootstrap enabled, the quality measure criterion used was Gini Impurity, as discussed in Section 4.7.1.

The proposed random search used a 4-fold cross validation technique before training a model with a randomly chosen hyperparameter combination. Thus, the original training data was divided into 4 subsets by arbitrarily sampling the data points (without replacement) using stratified k-fold cross validation. The model was then trained using *k-1= 4-1 = 3* subsets, with 1 validation subset for model evaluation (Berrar, 2024). Including the search space and *cv=4*, a random search instance `rf_rand_search` was created which selected 6 combinations of hyperparameters using `RandomizedSearchCV`. In 4-fold cross validation with 6 iterations, the training data was split into 4 folds to train and validate a model 4 times, with 6 different combinations of learning and evaluation. In other words, the algorithm learned $6 \times 4 = 24$ times from the training data using the `fit()` method (See Appendix F).

Hyper-parameters	Description	Default	Search Space
n_estimators	No. of trees in the forest	100	100 to 500, 5 outputs
max_features	No. of features	sqrt	sqrt
max_depth	Maximum depth	None	10 to 50, 5 outputs
min_samples_split	Minimum samples to split	2	5, 10, 15, 3 types
min_samples_leaf	Minimum samples at leaf node	1	2, 5, 10, 3 types
Bootstrap		True	True
Criterion	Quality measure	Gini	Gini

Table 2: Search space for Hyperparameter tuning using Random Search for Random Forest Classifier

5.1.3 Experimental Hyperparameter Tuning

Stage 3 in Figure 1 shows that, by keeping the best hyperparameter as a reference for experiments, a function `rf_experiment()` was created to experiment the model with different numbers of trees with various depths (see Appendix G). The function took *n_estimators* and *max_depth* as parameters. With 2,420 features in the proposed dataset, the number of trees in the forest should ideally be "*at least an order of magnitude higher than the number of features in order to exhaustively explore all the feature space during the forest building*" (Titapiccolo, et al., 2013), which is 24,200 trees in this case. With an increasing number of trees, the model variance decreases, and generalization ability improves. However, as more trees are added to the ensemble, the generalization error approaches a limit instead of decreasing indefinitely (Breiman, 2001). To investigate the point of convergence, the number *n_estimator* was increased from 100 up to 20,000. The *max_depth* after which the generalization stabilizes was also investigated with depths from as low as 10 to 110.

Similar to Section 5.1.2, the `RandomForestClassifier()` was initiated, with the best hyperparameter values found from random search set as defaults, because their impact on the model would not be investigated in this proposal. Function `rf_experiment()` was called with different combinations of *n_estimators* and *max_depth* values, each combination generating a Random Forest model. Each model was validated using 15% of validation data, which was extracted in a similar way as Section 5.1.1. The models were eventually tested with the unseen test set.

5.2 Custom CNN

5.2.1 Hyperparameter Tuning

Stage 1 performed a hyperparameter tuning using DL frameworks like ***TensorFlow*** and ***Keras*** to explore a range of parameters, as shown in Figure 1. A `custom_hyp_tune()` function was created with the `hp` parameter from *Keras Tuner* to instantiate its hyperparameter object. The object helps to define a search space for the parameters that are to be tuned. Using `keras.models.Sequential()`, a new model was created by successively connecting the extracted feature map from VGG16, a flatten layer, a FC layer, and an output layer in a stack (Géron, 2019). The flatten layer converts VGG16's feature maps into a 1-D vector using the `Flatten()` method**.** Subsequently, a FC layer was added to the sequence and tuned with a range from 8 to 100 neural units, by keeping in mind that too many units may lead to overfitting. A higher number of units means more neurons end up learning the feature patterns. A tuned dropout layer with rates like 0.25 and 0.5 was added to investigate the impact on learning ability when a certain probability of units is deactivated. With these dropout rates, every neuron could either have 25% or 50% probability of being temporarily ignored during training (Géron, 2019). Finally, a wide spectrum of learning rates (0.0001 to 0.001) was explored using ADAM optimizer to find the optimal weight configuration that performs soundly on both training and validation sets (Sanni, et al., 2024).

Using the hyperparameter search space in Table 3, the 70% augmented training sample was trained. An early stop was implemented to halt the training process after 2 epochs if there was no improvement in the validation accuracy (Sanni, et al., 2024). The search space was explored using *Keras Tuner*'s `RandomSearch()` class to exploit its efficient search capacity for high dimensional image data. During the randomized search, it assembled 5 models with 5 different hyperparameter combinations, by setting

max_trial=5 and trained them for 5 epochs. With a batch size of 128, there was 87 iterations for 11,201 training samples using equation (7).

$$\textbf{iterations per epoch} = \frac{\textbf{sample size}}{\textbf{batch size}} \quad \textbf{(7)}$$

With increasing epoch, the validation loss starts to decrease. However, at some point, it may start to increase again, pointing towards overfitting (Géron, 2019). Thus, an early stopping callback was incorporated for the iterative training to stop after 2 further epochs if the validation loss did not further improve. Lastly, each model was validated using the 15% validation data, and the hyperparameter combination with the best validation accuracy was recorded (see Appendix H).

Random Search Hyperparameters	Search Space
dense_units	8 to 100, 20 outputs
dropout	0.25, 0.50
learning_rate	0.00001 to 0.01
activation	ReLU
kernel_regularizer	L2(0.01)
optimizer	ADAM
Training Hyperparameters	**Value**
max_trial	5
patience	2
epoch	5

Table 3: Hyperparameter space for Custom CNN using Random Search

5.2.2 Fine-tuning Experiments

Similar to the ML experiments in Section 5.1.3, the best performing hyperparameters from random search were used in Stage 2 to execute fine tuning experiments, as shown in Figure 1 (also see Appendix I). For the custom CNN experimental setup, the dropout rate and learning rates were regulated to find the optimum combination, as done by Sanni, et al. (2024). While they also tested the effect of batch size on training speed and accuracy, the batch size throughout the proposed research was kept constant. Additionally, the adjustments in no. of units in the FC layer is also proposed in this research to investigate the pattern learning ability with increasing number of neurons. The experiment was conducted using `custom_model_experiments()` function, with *neurons*, *dropout*, and *learning_rate* as positional parameters. A `train_and_evaluate()` function used different combinations of arguments as hyperparameters to execute the experiments using the same early stopping setup and no. of epoch as in Section 5.2.1. This function generated the training time, accuracy-loss values for training and validation per epoch, and the best validation score with the test accuracy of the model. Note that the hyperparameters from random search were not tested previously, but were done during the experimentation.

6 Experimental Analysis

The proposal explored different hyperparameter combinations for both RF and Custom CNN classifiers to find the optimal model. The hyperparameters from random search were used as baselines to tune each classifier's neighbouring hyperparameter combinations. The reason behind these experiments was to train the classifiers at a broader spectrum, as the search space for random search may have been too small to find the best parameters. It also helped to explain the effects of imposing different conditions on a model's performance. While some variables were kept constant as found from random search, others were adjusted. For example, in RF, only the number of trees and their depths were explored, but for the custom CNN, the influence of different no. of neurons in the FC layer, dropout, and learning rates were investigated.

6.1 Random Forest

The initial training using the default classifier gave a validation accuracy of 89%, with 65.3% test accuracy. When the same data were trained using optimization methods to find the best hyperparameter values, the model scored with a cross-validation accuracy of 88.8%, as shown in Table 4. For the research experiments, parameters like *max_features*, *min_samples_split*, *min_samples_leaf*, *bootstrap*, and *criterion* were kept the same as found from random search. However, based on two hyperparameters of RF, namely *n_estimators* and *max_depth*, the proposed testing scenarios were divided into four parts, totalling 16 trials.

Parameters	Default	Hyperparameter tuning
n_estimators	100	100
min_samples_split	2	5
min_samples_leaf	1	2
max_features	sqrt	sqrt
max_depth	None	30
criterion	Gini	Gini
bootstrap	True	True
Validation Accuracy	0.8901	0.8877

Table 4: Parameters of Default Random Search Classifier (middle column); Parameters found after Random search with given search-space (right column)

	Model Trial	N estimators	Max depth	Training Time/s	Validation Accuracy	Test Accuracy	Generalization Gap
Min samples split=5, min samples leaf=2, max features=sqrt, criterion= Gini, Bootstrap= True							
	Default	100	None	129	0.8901	0.6531	0.2370
Test 1	1	100	30	124	0.8935	0.6805	0.2130
	2	200		247	0.9239	0.7199	0.2040
	3	300		371	0.9270	0.7461	0.1809
	4	400		495	0.9306	0.7524	0.1782
	5	500		617 (8m 25s)	0.9369	0.7687	0.1682
Test 2	6	100	10	91 (1m 52s)	0.8581	0.7089	0.1492
	7		30	134	0.8934	0.6805	0.2129
	8		50	136	0.8961	0.6830	0.2131
	9		70	135	0.8960	0.6830	0.2130
	10		90	135	0.8961	0.6830	0.2131
	11		110	133	0.8961	0.6830	0.2131
Test 3	12	2000	10	1745 (29m)	0.9052	0.7886	0.1166
	13	5000		4373 (1h 12m)	0.9085	0.7908	0.1177
	14	15000		13173 (3h 39m)	0.9082	0.7955	0.1127
	15	20000		–	–	–	–
T4	16	10000	10	13709 (3h 48m)	0.9070	0.7964	0.1106

Table 5: Experimental results for Random Forest Hyperparameter combinations

Test 1: For trial 1 to 5, the values of *n_estimators* were increased up to 500 to see the effect of increasing trees in a forest. The maximum depth of the tree was kept constant as the *max_depth=30* from the hyperparameter tuning in Table 4.

Test 1 Results: While the best cross validation accuracy from random search was with 100 trees (the lowest bound in the search space), when *n_estimators* is increased, the validation and test accuracies of each experimental model also rises. The training time also gets higher. Additionally, there was an observable decrease in the generalization gap. Table 5 shows that when the best hyperparameter values were evaluated again in trial 1, the validation accuracy improved by 0.0024, with 68% test accuracy. trial 5 used 500 estimators and recorded the highest validation accuracy of 94% with 77% test accuracy, making it the best model in Test 1.

Test 2: Trial 6 to 11 was carried out using a fixed number of 100 estimators, which was the same as the best hyperparameter values from random search. But the *max_depth* was gradually adjusted between 10 and 110 (inclusive) with an increment of 20.

Test 2 Results: With a depth of 10, trial 6 had a lower validation accuracy. The test accuracy was slightly higher than the rest of the group. However, as a tree went deeper, there was not much change of training times and accuracy results. Result in Table 5 suggests that the accuracy was enclosed around the depth of 30 and beyond that, there would be no significant improvement (Nadi & Moradi, 2019). In fact, lowering the depth to 10 resulted in better test accuracy. Because of having higher test reliability (71%) and smaller generalization gap, trial 6 was voted as the best model in Test 2.

Test 3: The last three experiments investigated the impact of *n_estimators > 500*. To exhaustively explore the entire feature space, the number of trees was recommended to be at least 24,200 by Titapiccolo, et al. (2013). Thus, much higher values like 2,000, 5,000, 15,000, and 20,000 were used. The *max_depth=10* was chosen because of its shorter training time to balance out the increasing time that may arise because of a higher number of trees.

Test 3 Results: The results of trial 12, 13, and 14 show that, with a dramatic increase of trees, the training time got much bigger. While the validation accuracy improved slightly, the test accuracy improved quite well. However, the improvement stabilised after a certain number of trees. For instance, 5,000 estimators had much better performance than 100; but when increased further, there was no significant improvement in the generalization gap. Furthermore, the training time was nearly 4 hours for 15,000 trees, which would be much higher for 20,000; thus, this training was not concluded. The model with 15,000 trees had the highest scores of 91% validation accuracy and 79.6% test accuracy, having the smallest generalization gap.

Test 4: The generalization Gap between 5,000 and 15,000 estimators was very low with almost similar accuracy scores. Hence, to get a model with something in the middle, trial 16 used 10,000 estimators, with the same configuration as Test 3.

Test 4 Results: The accuracies and training time were similar to the results with 15,000 estimators. However, the generalization gap slightly improved.

Since **trial 16** had the smallest generalization gap and highest test accuracy, the best hyperparameter combinations for RF classifiers was using *n_estimators*=10,000 with *max_depth*=10. This model was used for comparative analysis in Section 7.1.

6.2 Custom CNN

The best hyperparameter combination from random search suggested that with 28 dense units and 0.5 dropout rate, the model achieved 97% validation accuracy when the learning rate (LR) was 0.0007. Using these values as a starting point, different combinations were tested with seven experiments, having 26 trials. The *activation_function*, *kernel_regularizer*, and *optimizer* were fixed hyperparameters as shown in Table 6. For test 1 and 2, the LR was fixed to 0.0007 because it yielded a promising validation performance during the random search phase. From trial 1 to 25, models were trained for 5 epochs.

	Model Trial	Units	Dropout	Training Time/s	Training Accuracy	Validation Accuracy	Test Accuracy	Generalization Gap
Activation=ReLU, kernel_regularizer=L2(0.01), Optimizer=ADAM, Learning Rate = 0. 0007								
Test 1	1	28	0.5	380	0.2344	1	0.2801	0.7199
	2	64		382	0.3906	1	0.7278	0.2722
	3	128		382	0.4141	0.8911	0.7273	0.1638
	4	256		383	0.6250	0.9307	0.7273	0.2034
	5	512		383	0.5547	0.9010	0.7615	0.1395
	6	**1024**		**383**	**0.6250**	**0.9010**	**0.7896**	**0.1114**
Test 2	7	28	0.25	381	0.3082	0.4280	0.4086	0.0194
	8	64		380	0.5078	0.9802	0.7200	0.2602
	9	128		381	0.5625	0.8118	0.7485	0.0633
	10	256		382	0.6328	0.8713	0.7290	0.1423
	11	512		381	0.6719	0.8515	0.7490	0.1025
	12	**1024**		**382**	**0.7422**	**0.9208**	**0.7776**	**0.1432**

Table 6: Experimental Trials 1-12 for the Custom CNN

Test 1: After testing the hyperparameter combination from random search in trial 1, the neurons in the dense layer were increased exponentially for 5 terms, starting from 64. The dropout rate was fixed at 0.5 and LR was 0.0007 for trial 2-6.

Test 1 Results: Although the model from trial 1 had the highest validation accuracy 100% at epoch 4, the validation loss increased at epoch 5, indicating that the model was over-fitted on validation data. The training accuracy always remained below 25% with minimal improvement in the losses. This was because the model struggled to learn any meaningful patterns with a small number of units. On the other hand, 50% dropout allowed only 14 out of 28 neurons to actively learn the data. While the model learned the training data using only half the neurons, it validated using all the neurons. Also, the validation set was not augmented, unlike the training set, making it simpler to learn. Having a low test accuracy of 28%, the model exhibited inconsistent performance across all datasets, making it a very simple model with poor generalization ability.

For most cases in trial 2-6, the training and test accuracies went up with increasing no. of units, however there was a slight decline in the training accuracy using 512 units. The validation accuracy peaked to 93% using 256 units but decreased for the subsequent ones. With 1,024 units, the test accuracy was 78% which was the highest. Considering the lowest generalization gap of 0.1114 and highest training accuracy of 62.5%, Trial 6 performed the best with 1,024 units, 0.5 dropout and a LR of 0.0007.

Test 2: Using the same neural units and LR as Test 1, trial 7-12 investigated the impact of 25% dropout on model performance.

Test 2 results: Similar to Test 1, the training accuracy increased with more no. of units. In this case, the validation and test accuracies fluctuated till 512 neurons. For 1,024 neurons, training accuracy was the highest with 72%, and validation accuracy was 92% with a generalization gap of 0.1432. The test accuracy was 77.8%, suggesting that the model did not generalize well on unseen data and the model was overfitting on the validation set. Trial 12 had the highest training and test accuracies, with high validation accuracy. However, 72% training accuracy implies that the model did not fully capture the training data's complexity.

	Model Trial	Units	Dropout	Learning Rate	Training Time/s	Training Accuracy	Validation Accuracy	Test Accuracy	Generalization Gap
Test 3	15	1024	0.5	0.7	381	0.0846	0.1402	0.0657	0.0745
	14			0.07	382	0.1042	0.1246	0.1193	0.0053
	13			0.007	382	0.3125	0.8218	0.2851	0.5367
	16			**0.00007**	**381**	**0.7109**	**0.9010**	**0.8288**	**0.0722**
Test 4	17	1024	0.25	0.7	381	0.1136	0.1306	0.0233	0.1073
	19			0.07	381	0.0831	0.2092	0.1193	0.0899
	18			0.007	381	0.2422	0.9406	0.4638	0.4768
	20			**0.00007**	**381**	**0.7656**	**0.9010**	**0.8105**	**0.0905**
Test 5	21	2048	0.5	0.00007	384	0.7471	0.8433	0.8370	0.0063
	22		0.25	0.00007	382	0.6797	0.8515	0.8321	0.0194
	23		0.5	0.0007	381	0.6503	0.8025	0.7880	0.0145
	24		0.25	0.0007	382	0.7188	0.7921	0.7889	0.0032

Table 7: Experimental Trials 13-24 for the Custom CNN

Test 3: In both Test 1 and 2, the best performing models contained 1,024 units. Thus, for trial 13-16, the impact of different LR (except 0.0007) with 1,024 neurons were investigated. The LR were scaled up by a factor of 10 and the dropout was fixed at 50%. The results were recorded in descending order of learning rate in Table 7.

Test 3 Results: All the accuracies increased with decreasing LR. With the smallest LR of 0.00007, trial 16 had the highest performance of 71% training accuracy, 90% validation accuracy, and 83% test accuracy.

Even so, the model under-fitted on training data and over-fitted on validation data. Although the generalization gap was the smallest for LR=0.07, all the accuracy scores were below 12.5%. For LR=0.007, the model did not learn any significant pattern from training data and overfitted on validation data, with high generalization gap.

Test 4: Same as Test 3, but with 25% dropout for trial 17-20.

Test 4 Results: Although the increase in accuracy followed the same trend as Test 3 results, there was a 6.6% improvement in training accuracy, because more neurons were enabled to learn from training data. LR=0.007 also had similar performance as described in Test 3.

Test 5: A higher neural unit of 2,048 was used. Because of better performance with lower LR, the rate was fixed to 0.00007 for trials 21-22 with 50% and 25% dropouts respectively. In trial 23 and 24 tested for LR =0.0007.

Test 5 Results: For 0.00007, trial 21 learned better using 50% dropout rate compared to dropout=25%, and the generalization gap was very small with 84% validation accuracy. Therefore, increasing the units helped to reduce overfitting on validation data. Which means the model captured overall patterns from training data which generalised well on unseen data. LR =0.0007 portrayed a similar relationship, but with lower accuracy scores.

	Model Trial	Training Time/s	Epoch	Training Accuracy	Validation Accuracy	Test Accuracy	Generalization Gap
Dropout = 0, Learning Rate = 0.00007, units = 1024							
Test 6	25	465	5	0.7344	0.9208	0.8182	0.1026
Test 7	26	628	10	0.8281	0.9307	0.8465	0.0842

Table 8: Experimental Trials 25 & 26 to improve training accuracy with 0% dropout

Test 6: With lower dropout rates, the models demonstrated better learning from the training data**.** However, the previous experiments clearly showed that the models under-fitted when trained with dropout rates. Thus, trial 25 was tested with 1,024 units using LR =0.00007 because this combination showed best performance so far. Except, the dropout was removed by setting it to 0 to compare with the same configurations of other dropout cases.

Test 6 Results: The results showed a similar pattern as the trial 16 and 20 (with dropouts enabled). However, the validation accuracy improved; but at the same time the generalization gap increased, hinting to overfit. It was also observed that the model learned the training data less accurately compared to trial 20. The accuracy scores suggest that the model was under-fitted on training data.

Test 7: The final experiment was to increase the number of epochs to 10 to increase training time and analyse how well the model can improve if allowed to learn for a longer time (See Appendix J).

Test 7 Results: While the validation accuracy only increased by 1%, the training accuracy improved to 83% because of learning the patterns from data more effectively for a longer time. The test accuracy also increased by about 3%. The training and test accuracies were very close in this model, suggesting that the

original model performs almost similarly on unseen data. Figure 6 demonstrates that for both training and validation, the accuracies increased, and losses decreased over the epochs, indicating that the model learned the training data successfully and generalised well on unseen data. However, the validation accuracy increased erratically with the training time, suggesting that the model improved over time but struggled to generalize on certain patterns that may be present on unseen data. The training losses were higher than validation losses, which suggests that the model performed better by predicting easily using the validation data. This was because the validation data did not have data augmentations, and the training process used regularization to prevent overfitting.

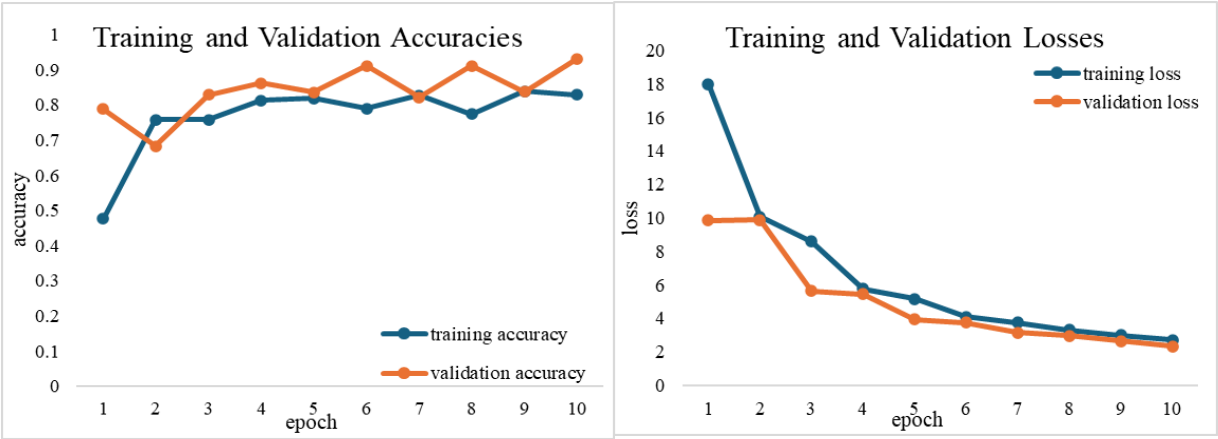


Figure 6: Training and validation accuracy (left) and loss (right) per epoch

From these experiments it can be concluded that 1,024 neurons had the best performance with a LR of 0.0007 compared to the rest of the trials. Furthermore, **trial 26** demonstrated the best performance among the rest with the highest training and testing accuracies, thus it was chosen as the CNN classifier for comparative analysis.

Table 9 shows the best performing models and their hyperparameters that were used for comparative analysis.

Name	Model	Hyperparameters	
Custom CNN	Trial 26	Units = 1024 Learning Rate = 0.00007 Kernel Regularizer=L2(0.01)	Dropout = 0 Activation=ReLU Optimizer=ADAM
Random Forest	Trial 16	N_estimators = 10,000 Min samples split=5, Max features=sqrt, Bootstrap= True	Max_depth = 10 Min samples leaf=2, Criterion= Gini,

Table 9: Hyperparameters of the finalized models

7 Classification Results and Discussion

7.1 Model Comparison

Table 10 shows that the custom CNN classifier not only trained significantly faster, but also had better overall performance compared to the RF classifier. The Random Forest achieved very high training accuracy of 99.57%, which is an expected behaviour by ensemble models (Google for Developers, 2024). On top of that, PCA reduced a substantial no. of features from the data, making it simpler to memorize with perfect accuracy. A lower validation accuracy of 90.70% suggests that the model had learned noises or very specific patterns from the training data. Therefore, the model didn't learn to generalize perfectly to unseen data. Nevertheless, the drop in accuracy was not big, which implies that the model was not completely overfitting. This could be because both the sets were from the same data distribution, as described in Section 4.2, sharing some similar feature characteristics. However, test accuracy was much lower than the training accuracy (79.65%), confirming the doubts of overfitting issues.

Contrarily, the moderate training accuracy of 82.81% suggests that the CNN did not fully memorise the training data but only learned some meaningful patterns. Since the training data was augmented, it was exposed to extra variations which helped the model to generalize on spatial features. The test accuracy was 84.65%, which was very close to the training accuracy and had a small generalization gap. This suggests that the model would perform fairly well on unseen data in the real world. The validation accuracy was slightly higher than the training accuracy because the learned features were evaluated against simpler and unaugmented data during training, which resulted in easier generalization. This model did not show significant over-fitting issues.

Classifier	Training time/s	Training Accuracy	Validation Accuracy	Test Accuracy
Custom CNN	628	82.81%	93.07%	84.65%
Random Forest	13709 (3h 48m)	99.57%	90.70%	79.65%

Table 10: Comparison of the best CNN and ML models

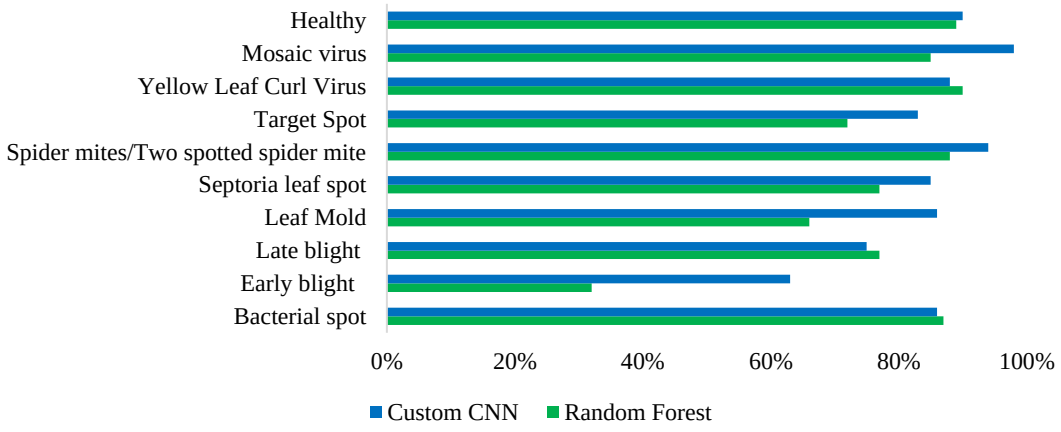


Figure 7: Model Performance Breakdown by Class

Figure 7 demonstrates that the CNN model predicted true positive cases more often than the RF classifier, especially for minority classes like Early Blights and Leaf Molds where RF underperformed significantly. Appendix C shows that these two classes (index 1 and 4) had overlapping features. Which means the synthetic samples created by SMOTE did not introduce enough variation in the data, thus the classifier could not differentiate between the high amounts of similar features. Since the aim was to build a disease detection system, the application demanded a higher rate of correct detection in the minority classes to accurately detect rare diseases (Chawla, et al., 2002), which the CNN managed to do but RF failed. However, another minority class Mosaic virus had very high recall for both models, because it had distinct features that did not overlap with other classes, hence SMOTE worked for this class. For majority classes like Yellow Leaf Curl Virus, Late Blight, and Bacterial Spots, RF performed marginally better, suggesting that the PCA successfully extracted enough features from these classes and learned diverse features. The close recall values of both models for Healthy, Yellow Leaf Curl Virus, Late Blight, and Bacterial Spot imply that these classes had well separated feature patterns, which can also be seen in Appendix C. Therefore, these classes did not require complex feature learning from synthetic samples because they all had enough data with distinguished features. Overall, the custom CNN model had higher recall values in seven out of ten classes, suggesting that it managed to learn more characteristics that would generalize better in real time. Table 11 shows the number of instances of each class in the data and the times the model predicted as True (both TP and TN). These results were documented using the confusion matrices (see Appendix K) and classification reports of both the models. Figure 8 visualizes the overall index results of both the models' performance metrics from their classification reports, where CNN consistently accomplished better. The CNN model had higher recall than RF, implying that the model had overall better sensitivity in detecting positive instances. It also had better precision compared to RF, thus predicting more true positive instances among the total positive instances. Considering the rationales discussed so far, the CNN model was chosen to be the better classifier for the diagnostic web application because of higher reliability at identifying more correct instances in real world scenarios.

Categories	Total Test	Correct	Sensitivity	Total Test	Correct	Sensitivity
	Random Forest			Custom CNN		
Bacterial spot	426	371	87%	320	274	86%
Early blight	200	38	32%	150	94	63%
Late blight	350	256	77%	287	215	75%
Leaf Mold	191	122	66%	143	123	86%
Septoria leaf spot	355	263	77%	266	225	85%
Spider mites/Two spotted spider mite	336	293	88%	252	236	94%
Target Spot	281	175	72%	211	175	83%
Yellow Leaf Curl Virus	642	593	90%	481	423	88%
Mosaic virus	75	53	85%	56	55	98%
Healthy	318	276	89%	239	216	90%

Table 11: Test results of each category for Random Forest (left) and custom CNN model (right)

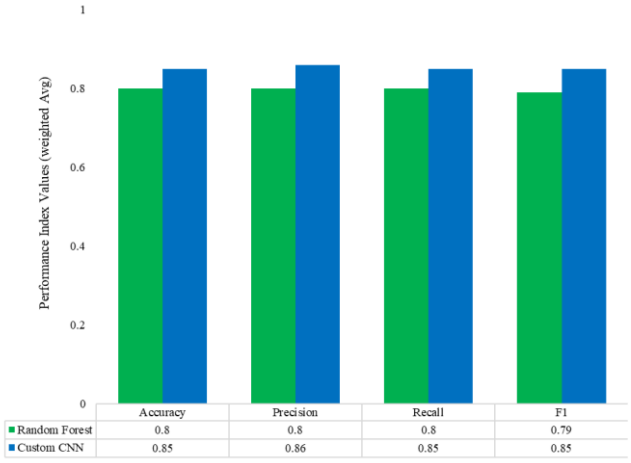


Figure 8: Classification report

7.2 Comparison with Existing Work

Study	Models	No. of Images	Classes	Params/ million	Subset	Test Accuracy
Proposed Model	Custom CNN	16,012	10	40.42	Tomato	84.66%
Durmuş, et al. (2017)	AlexNet	50,000 approx.	10	unknown	Tomato	95.65%
	SqueezeNet			unknown		94.30%
Hu, et al. (2020)	Improved GoogleNet	4354	4	5	Corn	97.60%
Islam, et al. (2023)	ResNet50	6000	10	25.6	Pepper, Potato, Tomato	98.98%
	VGG19			143.7		96.15%
	VGG16			138.3		92.39%

Table 12: Performance comparison of with existing related research using PlantVillage dataset

Table 12 presents a comparative survey of the proposed CNN model with the relevant literatures from Section 3. The analysis provides a broad perspective for assessing how well the proposed model performs relative to other existing benchmark methods that incorporate PlantVillage dataset. This is to identify strengths and weaknesses of the proposed model to gather insights for further improvements.

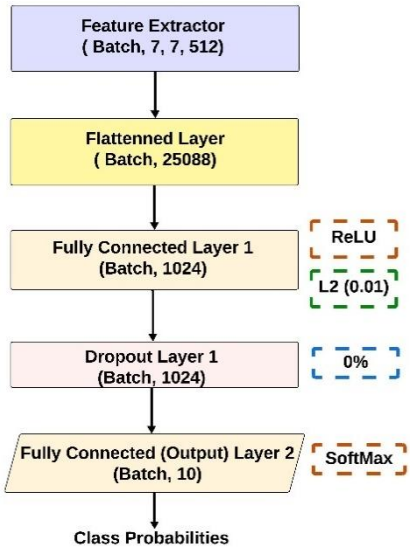


Figure 9: Architecture of the proposed custom CNN classifier

To build the tomato leaf disease classifiers, Durmuş, et al. (2017) used two complete CNN architectures to train more than 50,000 images from 10 different classes, and achieved more than 95% classification accuracy. The accuracy was much more than the proposed architecture, which could be due to the fact that they used about three times more data compared to the proposed dataset of 16,012 images. With more data, a model can learn more diverse feature patterns and generalize better on unseen data. Hu, et al. (2020) worked with 4354 corn leaf images from 4 categories of the same dataset and achieved 97.60% test accuracy. Although they used about one fourth of the proposed data size to classify fewer diseases, their architecture was much deeper than the proposed custom CNN model, as shown in Figure 9 (also see Appendix L). Their improved model consisted of 2 convolutional layers and 9 Inception modules. Each module contains multiple convolutional layers in parallel which can capture more high-level comprehensive features simultaneously compared to the the only 13 convolutional layers of VGG16 in the proposed feature extractor. Hence, learning more distinct feature patterns gave their model the advantage of higher classification accuracy with better generalization. Furthermore, the model was trained for 100 epochs, allowing it to learn the feature patterns for longer; whereas the proposed model was trained only for 10 epochs; thus, had less time to learn effectively. Additionally, their improved model was 8 times lighter than the proposed custom CNN model, making it more resource efficient. All three classifiers by Islam, et al. (2023) scored with much higher accuracies compared to the proposed CNN. Their best model had significantly deep architecture with 48 convolutional layers, allowing it to learn more data information. They also used VGG16 like the proposed model; however, they used the complete VGG16 architecture including its large neural units in the dense layers. On the other hand, the proposed model removed the top layers of VGG16, and passed the flattened features to a custom dense layer with less no. of units. Due to less neural units, the proposed model may not have learned sophisticated features as much, thus generalising less on unseen data. From these observations, it can be justified that the proposed custom CNN model underperformed because of having a simpler architecture, and it has room for improvements.

8 Web Application

The last stage of the proposal consisted of the development of an end-user interface. The interface was a web-based application that could operate on a local host server. To create the backend of the application, Flask, a Python based Web Framework was used. This lightweight framework accepts client requests, such as uploading images using HTTP methods like POST. The backend handles the preprocessing of the uploaded file before performing the classification using the trained custom CNN model. The uploaded file was configured to be resized to 224 × 224 pixels to match the transfer learning's input dimension, as described in Section 4.4. The input data can have a variable array of pixel values between 0 and 255, thus the values were normalized to a range between 0 and 1 to maintain consistency, as described in Section 4.3. There were ten tomato leaf health categories, and each category was stored in numerical format in alphabetical order. For each of these categories, the name of disease (if present), details of the disease, symptoms, recommended symptoms, and hyperlink to relevant sites are stored in a cloud-based NoSQL database called MongoDB Atlas (see Appendix M). With the loaded CNN algorithm, the backend used the `.predict()` function from DL frameworks to make predictions on the processed input data.

To make the classifications, the image was passed through the layers of the custom network (Figure 9). The filters, weights and the activation functions of the model were applied to the uploaded image to extract its features, which was then combined with the FC layers to predict the category using probability distribution of the classes.

Once the prediction was made, Flask performed a search query to look up for the predicted label's name in the database. Once a match was found, the backend returned the document of the predicted label from the database in JSON format. The JSON response was received by JavaScript in the frontend, which dynamically updated the content of the webpage. At the end, the webpage displayed the results (see the Video Link in Appendix N). The simple layout of the webpage was created using HTML, and visually enhanced using CSS.

The details of all the technical information are listed in Appendix D.

Appendix N lists all the relevant links to Google Drive, including the **final custom model** (in .h5 format) multimedia demonstration (.mp4), testing samples used in the video etc.

9 Conclusions

This research proposed a comparative assessment to determine the effectiveness of two different types of classifiers in making accurate diagnosis for tomato leaf diseases. For the learning process, ten categories of tomato leaves were used from the PlantVillage dataset. The proposed architectures included Random Forest, and a custom-made Convolutional Neural Network, which were incorporated with a CNN based feature extractor. Using VGG16's five convolutional blocks, the hierarchical features such as texture, edges, lesions and other spatial features on the foliage were extracted using transfer learning. The original Tomato leaf subset was imbalanced, which was addressed using class balancing methods. SMOTE oversampling technique was employed for Random Forest, and class weighing technique using Inverse of number of samples was implemented for CNN to prioritize the minority classes during training. To lessen the training time using Random Forest, Principal Component Analysis was used to transform the features into a lower dimension, while retaining the most important information from the original features. To identify a reference point for hyperparameter tuning experiments, the classifiers used Random Search to optimize the models using different hyperparameter combinations from a defined search space. This search optimization is simple to implement and can explore the search space more efficiently compared to other search methods in high dimensional space (Mantovani, et al., 2015). However, to minimize the computational time, the search space was deliberately kept constrained and it is acknowledged that using a wider range may have identified more accurate combinations. Even so, for RF it took about 5 and half hours to find the optimum hyperparameter using only 24 fits. Experimental results showed that for RF, higher no. of trees in the forest improved the generalization ability of the classifier, however the improvement converged after a certain increase in trees. For the CNN classifier, it achieved the best performance with lower learning rates, and the learning accuracy improved with lower dropouts. The increase in dense units did show fluctuating improvement, but the accuracies generally improved with more neural units. After conducting several experiments using different combinations of hyperparameters, the customised CNN outperformed RF classifier, achieving higher no. of true positive instances per class compared to the latter. Furthermore, the CNN predicted the minority classes like Early Blights and Leaf Molds with higher sensitivity, but RF was unsuccessful to predict these classes truly. Overall, the custom CNN model performed better by more correct positive instances and had higher classification accuracy of 85% compared to the RF classifier, which achieved only about 80%. This could have happened because PCA may have distorted certain features that could help the model to learn to generalize. Upon comparing with several feature extraction methods from relevant literatures, it was observed that the proposed VGG16 feature extractor extracted fewer complex features compared to other deeper architectures. This caused the proposed CNN architecture to classify with lower accuracy compared to the benchmark models. When tested using unseen images on the web application, some classes gave false positive results. For example, the video in Appendix N shows that for two instances, images of Septoria Leaf Spot and Late Blights were misclassified as a Leaf Mold.

The custom CNN model created in this research can be enhanced using more dense layers with higher units of neurons. The regularization strength of L2 can also be explored by adjusting the coefficient to prevent overfitting but at the same time to allow the model to learn useful patterns. The generalization can be improved further by introducing additional augmentation techniques such as flipping and rotations, so the model can learn more variation. Moreover, using deeper CNN architectures as feature extractors can aid in extracting more distinctive feature details. To get more accurate feature representations, better image processing techniques and manual feature extraction can be explored in future studies.

The disease identification interface was proposed to be an application on local host. As a part of future improvement plans, the app can be hosted on a cloud infrastructure to launch it online for public access, specifically for farmers. Furthermore, the web application was compatible for PC use only, which is not

convenient for farmers to use flexibly on fields. Thus, a lightweight architecture can be trained and deployed in mobile applications with additional functionalities, such as camera accessibility for uploading leaf images.

References

Aggarwal, S., Bhatia, M., Madaan, R. & Pandey, H. M., 2021. Optimized Sequential model for Plant Recognition in Keras. *IOP Conf. Series: Materials Science and Engineering*.

Ahmad et al., 2021. Plant Disease Detection in Imbalanced Datasets. *IEEE Access,* Volume 9.

Ahmad, M., ABDULLAH, M., MOON, H. & HAN, D., 2021. Plant Disease Detection in Imbalanced Datasets Using Efficient Convolutional Neural Networks With Stepwise Transfer Learning. *Cooperative Research Program for Agriculture Science and Technology Development.*

Ali, J., Khan, R., Ahmad, N. & Maqsood, I., 2012. Random Forests and Decision Trees. *IJCSI International Journal of Computer Science Issues,* 9(5).

Al-Janabi, M. & Andras, P., 2017. *A Systematic Analysis of Random Forest Based Social Media Spam Classification.* Newcastle-Under-Lyme, 11th International Conference.

amitjha11, 2020. *plant-disease-detection-using-vgg16,* s.l.: GitHUb.

Anower, O., 2024. *Web Application to Indentify Diseases in Tomato Leaves using Machine Learning and CNN,* London: s.n.

Bakirarar, B. & Elhan, A. H., 2023. Class Weighting Technique to Deal with Imbalanced Class Problem in Machine Learning: Methodological Research. *Turkiye Klinikleri Journal of Biostatistics.*

Banerjee, D. et al., 2024. *A Unified Approach to Tomato Leaf Disease Recognition: CNN and Random Forest Integration.* Pune, s.n.

Berrar, D., 2024. Cross-validation. *Encyclopedia of Bioinformatics and Computational,* 2(6).

Bharadiya , J. P., 2023. A Tutorial on Principal Component Analysis for Dimensionality Reduction in Machine Learning. *International Journal of Innovative Science and Research Technology,* 8(5).

Breiman, L., 2001. Random Forests. *Machine Learning,* Volume 45, pp. 5-32.

Brownlee, J., 2020. *How to Use Weight Decay to Reduce Overfitting of Neural Network in Keras.* [Online] Available at: https://machinelearningmastery.com/how-to-reduce-overfitting-in-deep-learning-with-weight-regularization/

Carignan, C., 2023. *Key to Common Problems of Tomatoes - University of Maryland.* [Online] Available at: https://extension.umd.edu/resource/key-common-problems-tomatoes/

ccslearningacademy, 2024. *What is Data Augmentation? Techniques, Examples & Benefits.* [Online] Available at: https://www.ccslearningacademy.com/what-is-data-augmentation/

Chakraborty, A., Chakraborty, A., Sobhan, A. & Pathak, A., 2024. Deep Learning for Precision Agriculture: Detecting Tomato Leaf Diseases with VGG-16 Model. *International Journal of Computer Applications,* 186(19).

Chawla, N. V., Bowyer, K. W., Hall, L. . O. & Kegelmeyer, . W. P., 2002. SMOTE: Synthetic Minority Over-sampling Technique. *Journal of Artificial Intelligence Research,* Issue 16.

Depenbusch, L. et al., 2023. *Tomato pests and diseases in Bangladesh and India: farmers' management and potential economic gains from insect resistant varieties and integrated pest management.* [Online] Available at: https://www.tandfonline.com/doi/full/10.1080/09670874.2023.2252760#abstract

Durmuş, H., Güneş , E. O. & Kirci, M., 2017. *Disease Detection on the Leaves of the Tomato Plants by Using Deep Learning.* Fairfax, VA, USA, IEEE.

Emmanuel, T. O., 2019. *PlantVillage Dataset - Kaggle.* [Online] Available at: https://www.kaggle.com/datasets/emmarex/plantdisease

Fiagbe, R., 2021. *HIGH-DIMENSIONAL RANDOM FORESTS,* THE UNIVERSITY OF TEXAS AT EL PASO: Department of Mathematical Sciences, Faculty of the Graduate School.

Géron, A., 2019. *Hands-on Machine Learning with Scikit-Learn, Keras & TensorFlow.* 2nd ed. CA: O'Reilly Media, Inc..

Google for Developers, 2024. *Random forests.* [Online] Available at: https://developers.google.com/machine-learning/decision-forests/random-forests

He, J. & Cheng, M. X., 2021. Weighting Methods for Rare Event Identification From Imbalanced Datasets. *Data Mining and Management.*

Hughes, D. P. & Salathé , M., 2015. An open access repository of images on plant health to enable the development of mobile disease diagnostics. *ArXiv,* Volume 1511.08060.

Hu, R. et al., 2020. *The identification of corn leaf diseases based on transfer learning and data augmentation.* Beijing, 3rd International Conference on Computer Science and Software Engineering (CSSE'20).

Islam, M. M. et al., 2023. DeepCrop: Deep learning-based crop disease prediction with web application. *Journal of Agriculture and Food Research.*

Joseph, N., 2022. *ml-glossary - GitHub.* [Online] Available at: https://github.com/bfortuner/ml-glossary/blob/master/docs/glossary.rst

Kiliçarslan, S. & Celik, M., 2021. RSigELU: A nonlinear activation function for deep neural networks. *Expert Systems With Applications.*

Kundu, R., 2022. *F1 Score in Machine Learning: Intro & Calculation.* [Online] Available at: https://www.v7labs.com/blog/f1-score-guide

MadhuriSrinivas, 2021. *Plant-Leaf-Disease-Prediction,* s.l.: GitHub.

Mantovani, R. G. et al., 2015. *Effectiveness of Random Search in SVM hyper-parameter tuning.* s.l., IEEE Xplore.

Masykur, F., Adi, K. & Nurhayati, O. D., 2022. *Classification of Paddy Leaf Disease Using MobileNet Model.* s.l., International Conference on Computing, Engineering and Design (ICCED).

Md.M. Islam et al. , 2023. DeepCrop: Deep learning-based crop disease prediction with. *Journal of Agriculture and Food Research.*

M, S., N, S. K. & V, T. B., 2017. *Recognition of Diseases in Paddy Leaves Using kNN Classifier*. Karnataka, IEEE.

Nadi, A. & Moradi, H., 2019. Increasing the views and reducing the depth in random forest. *Expert Systems with Applications.*

Pardede, J., Sitohang, B., Akbar, S. & Khodra, L. M., 2021. Implementation of Transfer Learning Using VGG16 on Fruit Ripeness Detection. *International Journal of Intelligent Systems and Applications(IJISA).*

pawangfg, 2024. *VGG-16 | CNN model - GeeksforGeeks.* [Online] Available at: https://www.geeksforgeeks.org/vgg-16-cnn-model/

Prabavathy, K. et al., 2023. *Plant Leaf Disease Detection using Machine Learning.* Coimbatore, Second International Conference on Applied Artificial Intelligence and Computing (ICAAIC 2023).

Prasvita, D. S. & Herdiyeni , Y., 2013. MedLeaf: Mobile Application For Medicinal Plant Identification Based on Leaf Image. *International Journal on Advanced Science, Engineering and Information Technology* , 3(2).

Rahman, Z. & Hossain, M. E., 2014. Role of Agriculture in Economic Growth of Bangladesh: A VAR Approach. *Journal of Business Studies* , Volume 7.

Rigatti, S. J., 2017. Random Forest. *JOURNAL OF INSURANCE MEDICINE.*

Samarth, V., 2023. *What is Random Forest In Data Science and How Does it Work?.* [Online] Available at: https://emeritus.org/in/learn/data-science-random-forest/

Sanni, S. L. et al., 2024. *Classification of Tea Leaf Clone Using Custom Convolutional Neural Network Based on VGG-16.* Semarang, 7th International Conference on Informatics and Computational Sciences (ICICoS).

Sharma, S., Guleria, K., Tiwari, S. & Kumar, S., 2022. A deep learning based convolutional neural network model with VGG16 feature extractor for the detection of Alzheimer Disease using MRI scans. *Measurement: Sensors.*

Shijie, J., Peiyi, J. & Siping, H., 2017. *Automatic detection of tomato diseases and pests based on leaf images.* Jinan, 2017 Chinese Automation Congress (CAC).

Singh, D. M. et al., n.d. *Biology OF Solanum lycopersicum (TOMATO).* UNEP/GEF supported Phase II Capacity Building Project on Biosafety ed. s.l.:Ministry of Environment, Forest and CLimate Change, Government of India; and Indian Institute of Vegetable Research, Varanasi.

Titapiccolo, J. I. et al., 2013. Artificial intelligence models to stratify cardiovascular risk in incident hemodialysis patients. *Expert Systems with Applications.*

Velliangiri, S., Alagumuthukrishnan, S. & joseph, I. T. S., 2019. *A Review of Dimensionality Reduction Techniques for Efficient Computation.* Hyderabad, Elsevier B.V..

Xu, Y. & Goodacre, R., 2018. On Splitting Training and Validation Set: A Comparative Study of Cross-Validation, Bootstrap and Systematic Sampling for Estimating. *Journal of Analysis and Testing.*

Yang, H. et al., 2021. *A novel method for peanut variety identification and classification by Improved VGG16*, s.l.: Scientific Reports.

Appendix A

```
# VGG16 with pre-trained weights and without fully connected layer
vgg16_feature_extractor = VGG16(include_top=False,
                  input_shape=(SIZE, SIZE, 3),
                  weights='imagenet')

# loaded layers as non-trainable
for layer in vgg16_feature_extractor.layers:
    layer.trainable = False

vgg16_feature_extractor.summary()
```

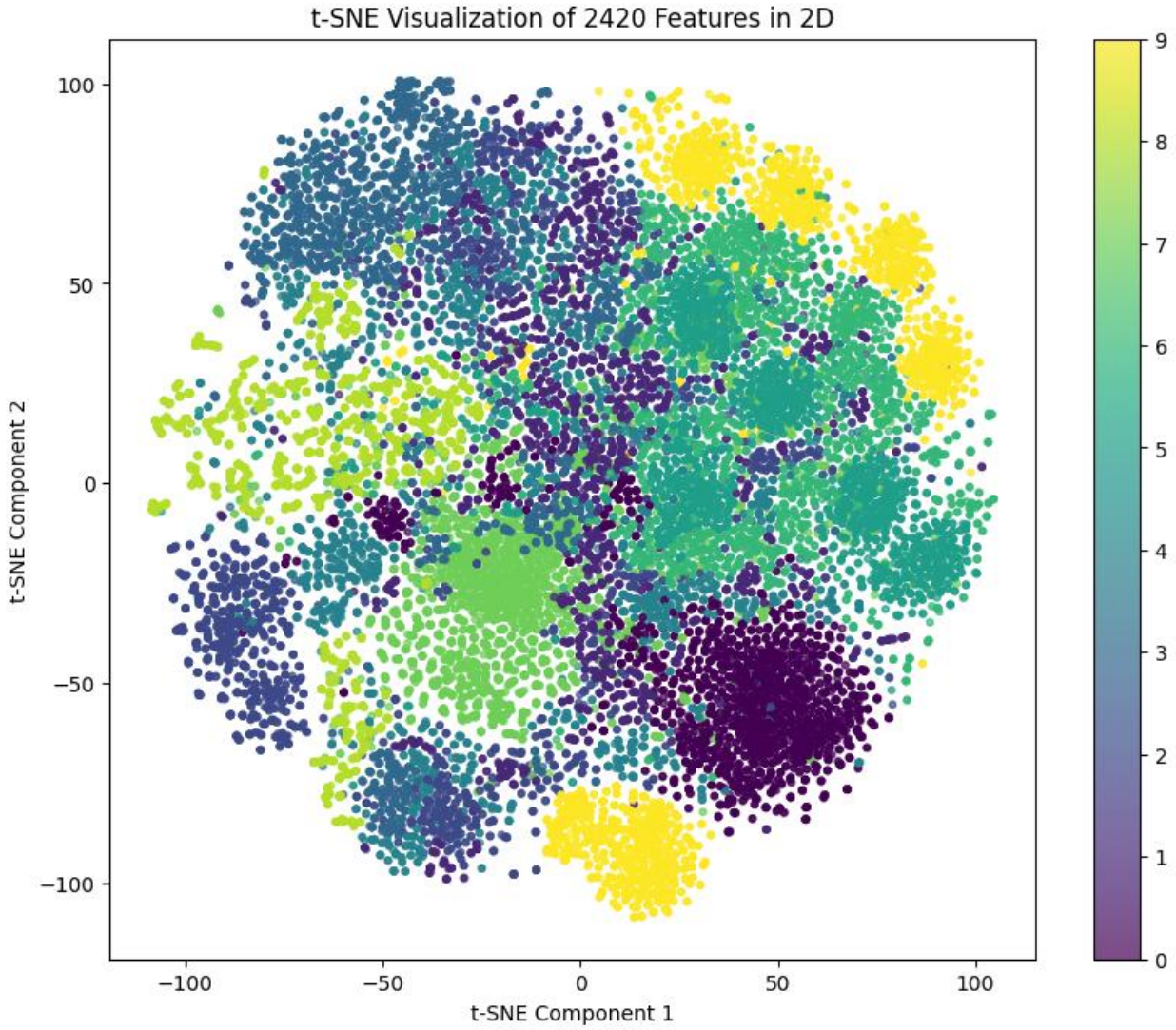
Appendix B

Model: "vgg16"

Layer (type)	Output Shape	Param #
input_layer (InputLayer)	(None, 224, 224, 3)	0
block1_conv1 (Conv2D)	(None, 224, 224, 64)	1,792
block1_conv2 (Conv2D)	(None, 224, 224, 64)	36,928
block1_pool (MaxPooling2D)	(None, 112, 112, 64)	0
block2_conv1 (Conv2D)	(None, 112, 112, 128)	73,856
block2_conv2 (Conv2D)	(None, 112, 112, 128)	147,584
block2_pool (MaxPooling2D)	(None, 56, 56, 128)	0
block3_conv1 (Conv2D)	(None, 56, 56, 256)	295,168
block3_conv2 (Conv2D)	(None, 56, 56, 256)	590,080
block3_conv3 (Conv2D)	(None, 56, 56, 256)	590,080
block3_pool (MaxPooling2D)	(None, 28, 28, 256)	0
block4_conv1 (Conv2D)	(None, 28, 28, 512)	1,180,160
block4_conv2 (Conv2D)	(None, 28, 28, 512)	2,359,808
block4_conv3 (Conv2D)	(None, 28, 28, 512)	2,359,808
block4_pool (MaxPooling2D)	(None, 14, 14, 512)	0
block5_conv1 (Conv2D)	(None, 14, 14, 512)	2,359,808
block5_conv2 (Conv2D)	(None, 14, 14, 512)	2,359,808
block5_conv3 (Conv2D)	(None, 14, 14, 512)	2,359,808
block5_pool (MaxPooling2D)	(None, 7, 7, 512)	0

Total params: 14,714,688 (56.13 MB)
Trainable params: 0 (0.00 B)
Non-trainable params: 14,714,688 (56.13 MB)

Appendix C


t-SNE Visualization of 2420 Features in 2D
t-SNE Component 2
t-SNE Component 1
100
50
0
−50
−100
−100
−50
0
50
100
9
8
7
6
5
4
3
2
1
0

Appendix D

Category	Machine Learning	CNN
	Details	
Dataset	Plant Village (Tomato)	
Architecture	Random Forest	Customized
Feature Extraction	VGG16 convolutional blocks	
Development Environment	Google Collab Pro	
Hardware	NVIDIA T4	NVIDIA L4
GPU	16 GB	22.5 GB
System RAM	25 GB	53 GB
Software	Python 3.11.11	
ML and DL Frameworks	TensorFlow, Keras API Scikit-learn, Imbalanced-learn	
Computer vision Frameworks	OpenCV	
Data Handling	Pandas	
Visualization	Seaborn, Matplotlib	
Model Persistence	Joblib	
	Web Application	
Operating System	Windows 11	
IDE	Visual Studio Code	
Software	Python	
Backend Framework	Flask	
Mark-up Language	HTML	
Content manipulation	JavaScript	
Style-Sheet Language	CSS	
Document-based Database	NoSQL (MongoDB Atlas)	

Appendix E

Train model with Default parameters

```
training_data, val_data, training_labels, val_labels = train_test_split(X_train_pcad,
                                                                        y_train_res,
                                                                        test_size=0.15,
                                                                        random_state=42,
                                                                        stratify=y_train_res)

rf_classifier = RandomForestClassifier(random_state=42)  # default rf

start = time.time()
rf_classifier.fit(training_data, training_labels)
end = time.time()
print(f"Training Time: {end - start}")
train_pred = rf_classifier.predict(training_data)
train_accuracy = accuracy_score(training_labels, train_pred)
print(f"Training Accuracy: {train_accuracy}")

val_pred = rf_classifier.predict(val_data)
val_accuracy = accuracy_score(val_labels, val_pred)
print(f"Validation Accuracy: {val_accuracy}")

test_pred = rf_classifier.predict(X_test_pcad)
test_acc = accuracy_score(y_test_load, test_pred)
print(f"Test Accuracy: {test_acc}")
```

```
Training Time: 128.5628638267517
Training Accuracy: 1.0
Validation Accuracy: 0.8901013250194856
Test Accuracy: 0.6531190926275993
```

Appendix F

Hyperparameter tuning to decide a search space for experiments

```
# https://medium.com/@prabowoyogawicaksana/hyperparameter-optimization-random-forest-classifier-550fd5ed8e14

hyp_param_rf = {
    'n_estimators': [int(x) for x in np.linspace(start=100, stop=500, num=5)],
    'max_features': ['sqrt','log2', None],
    'max_depth': [int(x) for x in np.linspace(10, 50, 5)],
    'min_samples_split': [5, 10, 15],
    'min_samples_leaf': [2, 5, 10],
    'bootstrap': [True],
    'criterion':['gini']
}

# stratified K-Fold cross-validator
strat_kfold_rf = StratifiedKFold(n_splits=4, shuffle=True, random_state=42)

rf_model = RandomForestClassifier()
rf_rand_search = RandomizedSearchCV(estimator=rf_model,
                                    param_distributions=hyp_param_rf,
                                    n_iter=6,
                                    cv=strat_kfold_rf,
                                    random_state=42,
                                    n_jobs=5, # uses 5 out of 8 CPUs
                                    verbose=2)

start_time = time.time()
rf_rand_search.fit(X_train_pcad, y_train_res)
end_time = time.time()
print(f"Time taken to Train the best RF mode using Random Seacrch: {end_time - start_time} seconds")

print(f"Best Hyperparameter: {rf_rand_search.best_params_}")
print(f"Best Cross-Validation Score: {rf_rand_search.best_score_}")

best_rf_model = rf_rand_search.best_estimator_

joblib.dump(best_rf_model, 'RandomForest_best_model.pkl')
files.download('RandomForest_best_model.pkl')
```

```
Fitting 4 folds for each of 6 candidates, totalling 24 fits
Time taken to Train the best RF mode using Random Seacrch: 19605.260795354843 seconds
Best Hyperparameter: {'n_estimators': 100, 'min_samples_split': 5, 'min_samples_leaf': 2, 'max_features': 'sqrt', 'max_depth': 30, 'criterion': 'gini', 'bootstrap': True}
Best Cross-Validation Score: 0.8877240841777085
```

Appendix G

Experiments

Uses same parameters found by random search.

only n_estimators and max depth are adjusted

```
def rf_experiment(n_estimators, max_depth, trial_id=1):

    training_data, val_data, training_labels, val_labels = train_test_split(X_train_pcad,
                                                                            y_train_res,
                                                                            test_size=0.15,
                                                                            random_state=42,
                                                                            stratify=y_train_res)

    print(f"Training with: n_estimators={n_estimators}, max_depth={max_depth}")

    # other parameters are same as the best hyperparameters
    rf_classifier_ex = RandomForestClassifier(n_estimators=n_estimators,
                                              max_depth=max_depth,
                                              min_samples_split=5,
                                              min_samples_leaf=2,
                                              max_features='sqrt',
                                              bootstrap=True,
                                              criterion='gini',
                                              random_state=42)


    start = time.time()
    rf_classifier_ex.fit(training_data, training_labels)
    end = time.time()
    print(f"Training Time: {end - start}")

    # validation accuracy
    val_pred = rf_classifier_ex.predict(val_data)
    val_accuracy = accuracy_score(val_labels, val_pred)
    print(f"Validation Accuracy: {val_accuracy}")

    # test accuracy
    test_pred = rf_classifier_ex.predict(X_test_pcad)
    test_acc = accuracy_score(y_test_load, test_pred)
    print(f"Test Accuracy: {test_acc}")

    # trial model download
    model_filename = f"rf_model_trial_{trial_id}.pkl"
    joblib.dump(rf_classifier_ex, model_filename)

    return val_accuracy, test_acc, model_filename
```

Appendix H

```
def custom_hyp_tune(hp):

    custom_hp = keras.models.Sequential()
    custom_hp.add(vgg16_feature_extractor)
    custom_hp.add(Flatten())
    custom_hp.add(Dense(hp.Int('dense_units', min_value=8, max_value=100, step=20),
                        activation='relu',
                        kernel_regularizer=regularizers.l2(0.01)))
    custom_hp.add(Dropout(hp.Choice(name='dropout', values=[0.25, 0.5])))
    custom_hp.add(Dense(10, activation='softmax'))

    custom_hp.compile(optimizer=keras.optimizers.Adam(learning_rate=hp.Float('learning_rate',
                                                                             min_value=1e-5, max_value=1e-2,
                                                                             sampling='log')),
                      loss='categorical_crossentropy', metrics=['accuracy'])

    return custom_hp
```

```
# Hyperparameters
k_tuner = kt.RandomSearch(
    custom_hyp_tune,
    objective='val_accuracy',
    max_trials=5,
    seed=42,
    directory='keras_tuning_directory',
    project_name='hyperparameter_tuning'
)

early_stop = EarlyStopping(monitor='val_loss',
                           patience=2,
                           restore_best_weights=True)

cp = ModelCheckpoint(filepath='custom_best_hyperparameter_model.keras', verbose=1, save_best_only=True)

# Random Search
start = time.time()
k_tuner.search(train_gen,
               epochs=5,
               steps_per_epoch = train_samples // batch_size,
               validation_data=val_gen,
               validation_steps = val_samples // batch_size,
               class_weight = class_weights,
               verbose=1,
               callbacks=[cp, early_stop])
end = time.time()
print(f"Search time: {end - start} seconds")
```

```
Trial 5 Complete [00h 06m 25s]
val_accuracy: 0.3103298544883728

Best val_accuracy So Far: 0.9702970385551453
Total elapsed time: 00h 33m 13s
Search time: 1993.128294467926 seconds
```

```
best_hyperparameter = k_tuner.get_best_hyperparameters(1)[0]
best_hyperparameter.values
```

```
{'dense_units': 28, 'dropout': 0.5, 'learning_rate': 0.0007102715674987189}
```

Appendix I

```
def custom_model_experiments(neurons, dropout, learning_rate):

        custom_model = keras.models.Sequential()
        custom_model.add(vgg16_feature_extractor)
        custom_model.add(Flatten())
        custom_model.add(Dense(neurons, activation='relu', kernel_regularizer=regularizers.l2(0.01)))
        custom_model.add(Dropout(dropout))
        custom_model.add(Dense(10, activation='softmax'))

        custom_model.compile(optimizer=keras.optimizers.Adam(learning_rate=learning_rate),
                        loss='categorical_crossentropy', metrics=['accuracy'])

        return custom_model
```

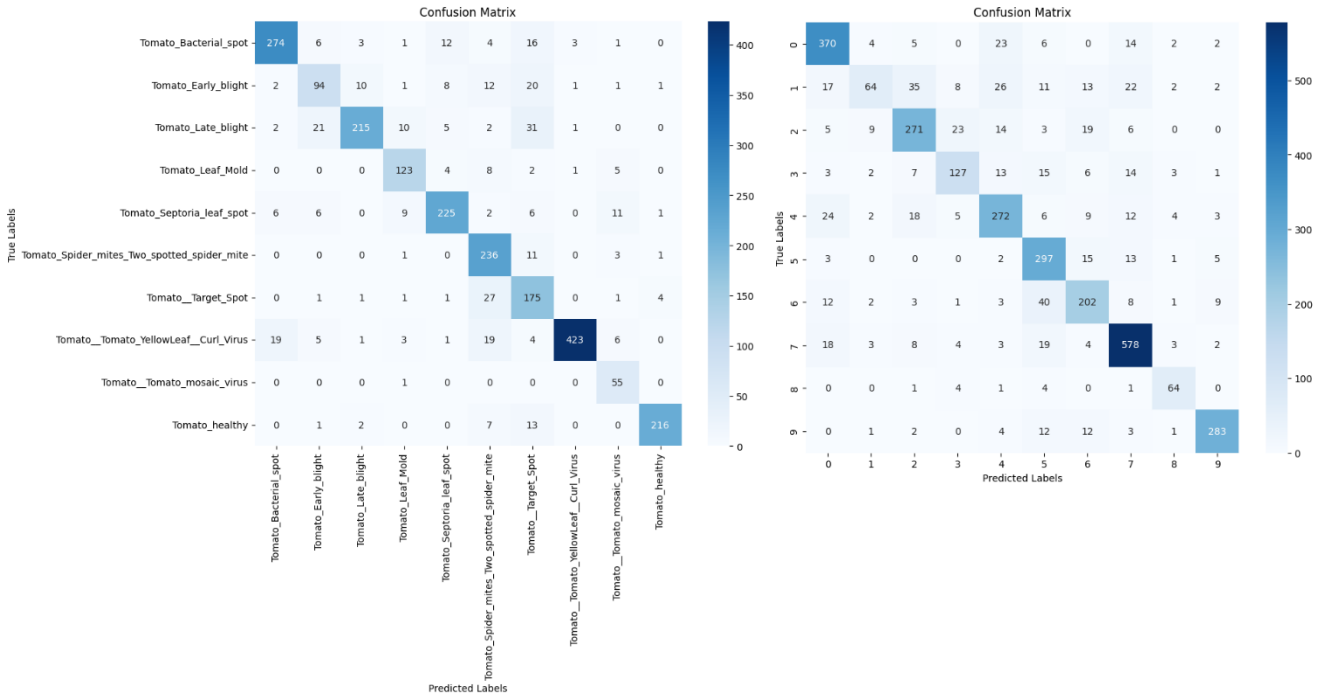
```
[ ]  def train_and_evaluate(neurons, dropout, learning_rate, trial_id=1):

        model = custom_model_experiments(neurons, dropout, learning_rate)

        early_stop = EarlyStopping(monitor='val_loss',
                                   patience=2,
                                   restore_best_weights=True,
                                   verbose=1)

        print(f"for neurons={neurons}, dropout={dropout}, learning_rate={learning_rate}:")

        start = time.time()
        best_model_history = model.fit(
            train_gen,
            steps_per_epoch = train_samples // batch_size,
            validation_data = val_gen,
            epochs = 5,
            validation_steps = val_samples // batch_size,
            class_weight = class_weights,
            callbacks=[early_stop],
            verbose=1
            )
        end = time.time()
        print(f"Training time: {end - start} seconds")

        val_accuracy = max(best_model_history.history['val_accuracy'])  # among 5 epoch
        print(f"Highest Validation Accuracy: {val_accuracy}")

        test_loss, test_accuracy = model.evaluate(test_gen, verbose=1)
        print(f"Test Accuracy: {test_accuracy}")

        model_filename = f'Experimental_trial_{trial_id}_model.h5'
        model.save(model_filename)
        print(f"Model for experimental trial {trial_id} saved as {model_filename}")

        return val_accuracy, test_accuracy, model_filename
```

Appendix J

```python
def train_and_evaluate_improve(neurons, dropout, learning_rate, trial_id=1):

    model = custom_model_experiments(neurons, dropout, learning_rate)

    print(f"for neurons={neurons}, dropout={dropout}, learning_rate={learning_rate}:")

    start = time.time()
    best_model_history = model.fit(
        train_gen,
        steps_per_epoch = train_samples // batch_size,
        validation_data = val_gen,
        epochs = 10,
        validation_steps = val_samples // batch_size,
        class_weight = class_weights,
        #callbacks=[early_stop],
        verbose=1
        )
    end = time.time()
    print(f"Training time: {end - start} seconds")

    val_accuracy = max(best_model_history.history['val_accuracy'])  # among 10 epoch
    print(f"Highest Validation Accuracy: {val_accuracy}")

    test_loss, test_accuracy = model.evaluate(test_gen, verbose=1)
    print(f"Test Accuracy: {test_accuracy}")

    model_filename = f'Experimental_trial_{trial_id}_model.h5'
    model.save(model_filename)
    print(f"Model for experimental trial {trial_id} saved as {model_filename}")

    return val_accuracy, test_accuracy, model_filename
```

Appendix K

Figure 10: Confusion Matrix of Custom CNN Model (left) and Random Forest Classifier (right)



Appendix L

```
[ ] final_model = load_model('/content/drive/My Drive/Experimental_trial_26_model.h5')
```

```
WARNING:absl:Compiled the loaded model, but the compiled metrics have yet to be built. `moc
```

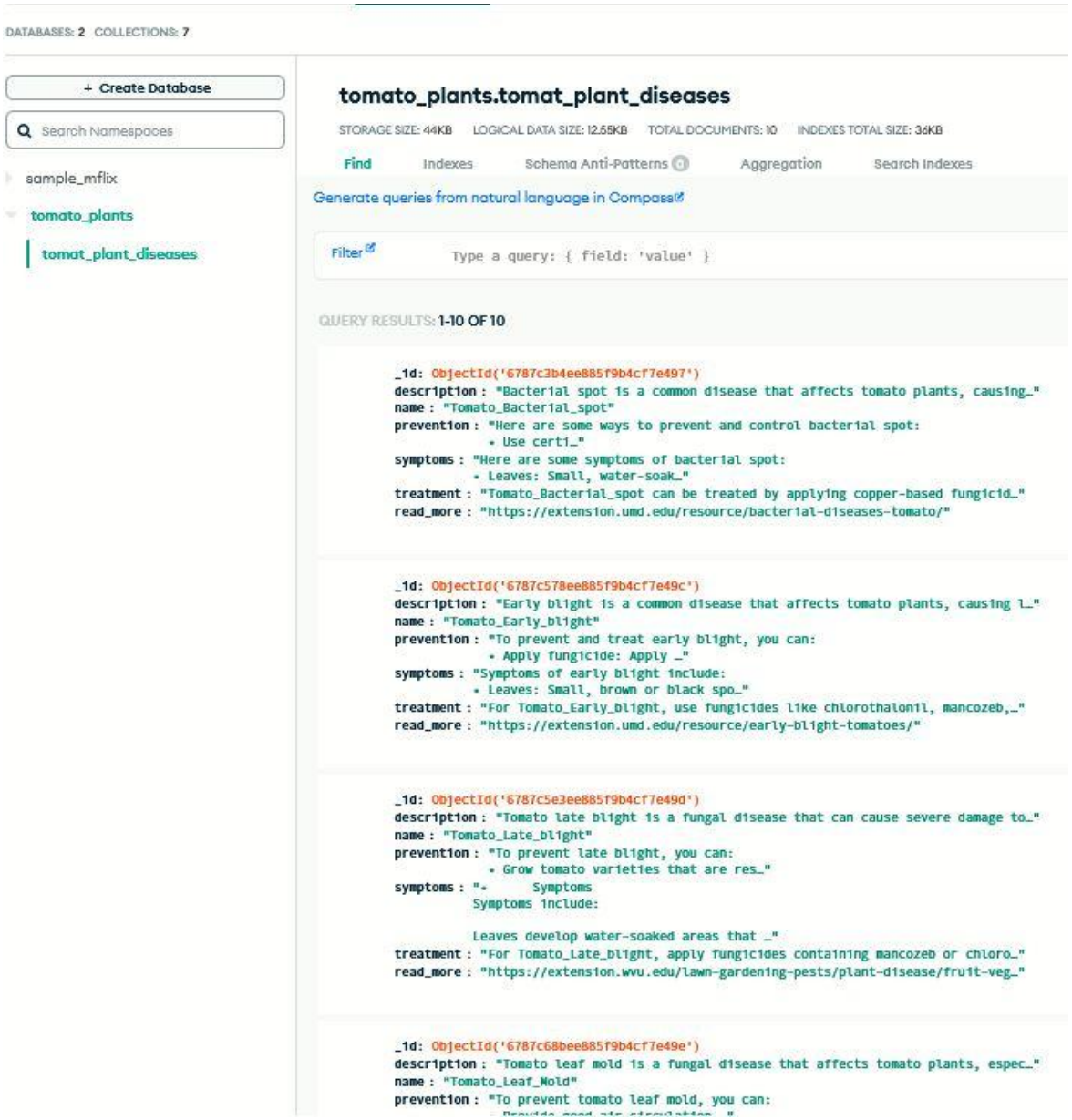
```
[ ] final_model.summary()
```

Model: "sequential_1"

Layer (type)	Output Shape	Param #
vgg16 (Functional)	(None, 7, 7, 512)	14,714,688
flatten_1 (Flatten)	(None, 25088)	0
dense_2 (Dense)	(None, 1024)	25,691,136
dropout_1 (Dropout)	(None, 1024)	0
dense_3 (Dense)	(None, 10)	10,250

Total params: 40,416,076 (154.18 MB)
Trainable params: 25,701,386 (98.04 MB)
Non-trainable params: 14,714,688 (56.13 MB)
Optimizer params: 2 (12.00 B)

Appendix M


DATABASES: 2 COLLECTIONS: 7
+ Create Database
Search Namespaces
sample_mflix
tomato_plants
tomat_plant_diseases
tomato_plants.tomat_plant_diseases
STORAGE SIZE: 44KB LOGICAL DATA SIZE: 12.55KB TOTAL DOCUMENTS: 10 INDEXES TOTAL SIZE: 36KB
Find Indexes Schema Anti-Patterns Aggregation Search Indexes
Generate queries from natural language in Compass
Filter
Type a query: { field: 'value' }
QUERY RESULTS: 1-10 OF 10
_id: ObjectId('6787c3b4ee885f9b4cf7e497')
description : "Bacterial spot is a common disease that affects tomato plants, causing…"
name : "Tomato_Bacterial_spot"
prevention : "Here are some ways to prevent and control bacterial spot:
• Use certi…"
symptoms : "Here are some symptoms of bacterial spot:
• Leaves: Small, water-soak…"
treatment : "Tomato_Bacterial_spot can be treated by applying copper-based fungicid…"
read_more : "https://extension.umd.edu/resource/bacterial-diseases-tomato/"
_id: ObjectId('6787c578ee885f9b4cf7e49c')
description : "Early blight is a common disease that affects tomato plants, causing l…"
name : "Tomato_Early_blight"
prevention : "To prevent and treat early blight, you can:
• Apply fungicide: Apply …"
symptoms : "Symptoms of early blight include:
• Leaves: Small, brown or black spo…"
treatment : "For Tomato_Early_blight, use fungicides like chlorothalonil, mancozeb,…"
read_more : "https://extension.umd.edu/resource/early-blight-tomatoes/"
_id: ObjectId('6787c5e3ee885f9b4cf7e49d')
description : "Tomato late blight is a fungal disease that can cause severe damage to…"
name : "Tomato_Late_blight"
prevention : "To prevent late blight, you can:
• Grow tomato varieties that are res…"
symptoms : "• Symptoms
Symptoms include:
Leaves develop water-soaked areas that …"
treatment : "For Tomato_Late_blight, apply fungicides containing mancozeb or chloro…"
read_more : "https://extension.wvu.edu/lawn-gardening-pests/plant-disease/fruit-veg…"
_id: ObjectId('6787c68bee885f9b4cf7e49e')
description : "Tomato leaf mold is a fungal disease that affects tomato plants, espec…"
name : "Tomato_Leaf_Mold"
prevention : "To prevent tomato leaf mold, you can:

Appendix N

VIDEO Demonstration Link (Google Drive)

https://drive.google.com/file/d/17TdJHJhcdUZnDX2sggJohNqJ3f-FTASo/view?usp=sharing

Test dataset (from CNN) Google Drive:

https://drive.google.com/drive/folders/1gBZuEj64Cfrr_Si6jMhaI-ZmrLfcJcE4?usp=sharing

Final Custom CNN model (Google Drive):

https://drive.google.com/file/d/1yzIuZDIhaPuLwcISBMQ7AjiPLC4O5Mso/view?usp=sharing

GitHub Repo:

https://github.com/Birkbeck/msc-projects-2023-4-0anower.git

Google Colab ML:

https://colab.research.google.com/drive/10PmYNgpjpLPk6FIkN4xmdzwk9m9Hm-so?usp=sharing

Google Colab CNN:

https://colab.research.google.com/drive/1TDousKvYr_4-dGnEgEkeZsdUotvLuwlz?usp=sharing